Desarrollo de un repositorio central de bloques para *Snap!*

Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya

Autor: Eduardo Delgado

Director: Bernat Romagosa

Ponente: Jordi Delgado

Fecha: 26/10/2016

Índice

Índice de figuras

Resumen

El objetivo de este proyecto de final de grado es el de construir un repositorio central de bloques, supervisado por el controlador de versiones *git*, para los usuarios de *Snap!*[1]. Dicho repositorio estará localizado en un servidor *http* que crearemos. Lo haremos accesible desde su plataforma web y desde una aplicación web propuesta. Los tres elementos (servidor + acceso desde la plataforma + aplicación web) definirán la interacción entre el usuario y el repositorio remoto.

1. **Introducción**

En este documento se describen y valoran los aspectos más relevantes del proyecto. Se comentan tanto la planificación temporal como la económica, y su resultado final. Se revisará la resolución de los objetivos marcados y se valorará la forma en la que ha cubierto el alcance que se había propuesto. Finalmente se comentarán las conclusiones a las que se ha llegado una vez se ha valorado el resultado final y se hará una valoración global y personal del proyecto.

El proyecto que se ha llevado a cabo es un trabajo de final de grado realizado en la Facultat d'Informàtica de Barcelona en la Universitat Politècnica de Catalunya en colaboración con el *Citilab*. Todo su contenido se enviará a los responsables de mantener y actualizar el sistema, para que sea añadido, tarde o temprano, al código oficial (*opensource*[2]) de *Snap!*.

1.1 Contexto

Snap! es un lenguaje de programación tipo *drag-and-drop* basado en *Scratch*[3] programable desde el navegador, que permite programar fácil y rápidamente, con mayor o menor complejidad, gracias a que la edición de código se hace simplemente moviendo y situando bloques, de forma adecuada, interactuando dentro del mismo navegador para su fácil acceso. Este software es totalmente libre y está situado en la nube, implementado en *javascript*.

En referencia al intercambio de bloques entre desarrolladores (código al fin y al cabo), el usuario puede descargar conjuntos de bloques como un archivo de texto en formato *xml.* El sistema permite cargar archivos que codifican conjuntos de bloques al entorno de trabajo de forma local. Asimismo, siguiendo el mismo procedimiento podemos exportar e importar proyectos enteros. A diferencia de los conjuntos de bloques, la plataforma sí que permite almacenar proyectos en la nube. Se puede intuir que, hasta el momento, la única forma que tienen dos usuarios de compartir un conjunto de bloques es descargando este archivo y enviándolo por correo electrónico, lo que implica que ambos han de estar en contacto.

El código es *opensource* por lo que está abierto a modificaciones y reimplementaciones.

1.2 Formulación del problema

La limitación de *Snap!* que ha querido superar este proyecto es que hasta el momento hay una gran distancia entre desarrolladores que evita el intercambio de bloques. En otras palabras, dos desarrolladores sólo podían compartir parte de su trabajo mediante el intercambio de archivos, lo que resulta imposible si ambos no están en contacto. Desde que la nueva herramienta se ponga en funcionamiento, todos los desarrolladores estarán siempre en contacto y podrán compartir código libremente y de forma prácticamente anónima.

Desarrollar un repositorio remoto desde cero implica también crear todas aquellas estructuras de las que dependerá, tanto para mantener su consistencia como para establecer puntos de acceso. Por otra parte, este factor nos proporciona más libertad en el momento de decidir la estructura que queremos que tenga, puesto que no partimos de una inacabada.

Los elementos de la estructura, que se detallarán más adelante, son, por una parte, un servidor *http* donde situaremos el repositorio y que se encargará de procesar las peticiones hacia el mismo y, por otra parte, dos puntos de acceso al mismo: uno desde la misma plataforma web y el otro desde una nueva aplicación web que construiremos, también desde cero.

2 Estado del arte

2.1 Actores implicados

El desarrollo/ampliación de una herramienta tan versátil como útil atrae el interés de numerosos usuarios finales de diferentes perfiles, así como supervisores externos que, en caso de adaptarse perfectamente a sus expectativas, validen e incluyan la ampliación a la versión oficial.

2.1.1 Director del proyecto

El director del proyecto será quien realice el seguimiento del trabajo. Gracias a su supervisión y experiencia con el lenguaje, este proyecto podrá cumplir con los objetivos que se ha propuesto.

2.1.2 Desarrolladores principales del software

Pese a ser una herramienta libre, siempre hay alguien detrás con la responsabilidad de que el software al que usuarios externos tienen acceso, sea totalmente funcional. El organismo siempre está abierto y fomenta la ampliación del proyecto para hacerlo más robusto y útil pero han de asegurarse de que aquella ampliación que quiere acoplarse al código 'oficial' esté libre de errores y sea lo suficientemente importante como para ofrecerlo como una actualización. Como que la funcionalidad desarrollada en este trabajo es lo suficientemente considerable a la par que esperada por la comunidad de usuarios que utilizan Snap!, lo más probable es que presten atención al resultado final para revisarlo y compararlo minuciosamente con el estándar exigido, siempre que este trabajo cumpla de forma excelente y adecuada, con todos los objetivos que se ha marcado.

2.1.3 Usuarios

El lenguaje de *Snap!* es fácil de manejar, muy intuitivo y de código abierto. Con estas características, siempre basadas en *Scratch*, podemos dividir a los usuarios que le sacarán más provecho en dos grupos mayoritarios. Puesto que este trabajo le proporcionará una herramienta más, podemos suponer que el perfil de los usuarios que usarán esta nueva versión, será similar al que actualmente trabajan con él.

Por una parte, tenemos a esos usuarios que buscan un primer contacto con el mundo de la programación y encuentran en este tipo de lenguajes una opción de aprendizaje a más alto nivel, es decir, una opción de programar evitando el contacto directo con del código. Gracias a que otros desarrolladores tienen la opción de compartir proyectos enteros y, desde ahora bloques individuales gracias a esta actualización, podrán progresar rápidamente con programas simples, pero completos. El motivo por el cual este grupo de usuarios preferirían empezar con *Snap!* y no con *Scratch*, es que son programadores que aspiran a desarrollar proyectos con un mínimo de complejidad más

allá de simples juegos, por ejemplo, *Snap!* permite crear objetos que puede ordenar de forma jerárquica.

Por otra parte, los usuarios y desarrolladores más experimentados, probablemente exprimirán el código fuente proporcionado junto al concepto de programación que plantea *Snap!* para situarlo como base de lenguajes complejos, para ser manipulados con la idea *drag-and-drop* simplificando el futuro desarrollo de software. Como que, una vez este proyecto haya sido capaz de incluir la herramienta descrita anteriormente, los desarrolladores verán la base sobre la que partieron modificada, y pasarán a incorporarla a esa versión concreta que hayan desarrollado, siempre que les sea de utilidad. Por ejemplo, en el caso del *Citilab*, se está trabajando en la incorporación de *Snap!* en el entorno del dispositivo *Arduino* tratando de que sea programable con el lenguaje por bloques lo que implica que, nuevamente, su desarrollo sea más fácil, intuitivo y rápido. Ese ejemplo nos muestra que el concepto también puede trasladarse al mundo de los dispositivos a partir del código fuente de la versión estándar.

2.2 Estudio del mercado

2.2.1 *Scratch*

Es un lenguaje de programación visual por bloques orientado a usuarios que buscan un primer contacto con el mundo de la programación. Es el primer elemento que hemos de analizar sencillamente porque *Snap!* nació como una reimplementación de este lenguaje. *Scratch* está orientado a la programación de juegos muy simples por medio de bloques (como *Snap!,* lógicamente). Por limitaciones como esas, se planteó la posibilidad de ir más allá de las posibilidades que ofrecía *Scratch*. Se buscó explotar la idea de la programación por bloques dando un giro al objetivo del lenguaje, orientándolo a un tipo de usuario más experimentado que no busca únicamente la implementación de juegos simples.

2.2.2 Lenguajes parecidos a *Snap!*

Grupos de desarrolladores han desarrollado y desarrollan expansiones de la plataforma con el objetivo de reducir al máximo la necesidad de programar escribiendo código. De la combinación de la facilidad que ofrece la programación por bloques y la

complejidad de entornos de programación concretos, puede surgir un nuevo lenguaje visual por bloques con muchas de las funcionalidades que ofrecería su programación directamente mediante código.

El lenguaje es *opensource* y propone que desarrolladores creen su propio lenguaje basado en *Snap!.* Las nuevas extensiones/reimplementación suelen acotar su entorno explotando aquellas partes que les resulten más útiles para su nuevo lenguaje. Puesto que su base parte de *Snap!* no dispondrán de un repositorio central de bloques, a no ser que hayan desarrollado uno por su cuenta. En ese caso, lo más probable es que ese repositorio esté muy adaptado a su nuevo lenguaje por lo que no sería fácil exportarlo a otras reimplementaciones, en otras palabras, seguramente no pueda ser compartido con las demás. Si por lo contrario definimos el acceso al repositorio central de bloques directamente en *Snap!*, cada reimplementación simplemente tendría que actualizar el código y seguir el estándar propuesto.

Podemos nombrar dos reimplementaciones desarrolladas en el *Citilab*:

- *Beetleblocks*[4]: Permite una programación gráfica en un entorno 3D
- *Snap4Arduino*[5]: Permite interaccionar con cualquier dispositivo *Arduino*

2.2.3 *App Inventor*[6]

El método de programación de *App Inventor* es parecido al que tienen otras expansiones de *Snap!*, solo que en este caso está centrado en el desarrollo de aplicaciones para *Android*. El resultado está muy bien conseguido: permite programar de forma muy intuitiva aplicaciones con un nivel considerable de complejidad totalmente funcionales y con una comunicación directa con un dispositivo o un emulador. Pero como en los casos anteriores, no dispone de un repositorio central de bloques sino que el usuario sólo tiene la posibilidad de importar/exportar proyectos completos.

2.2.4 *CPAN* y *meta::cpan*[7]

Es definitivamente un sistema comparable con el que se ha querido desarrollar en este proyecto. De hecho se podría decir que la intención del proyecto era acercarse a un resultado parecido a *meta::cpan*, orientado a *Snap!*. Para hablar sobre *meta::cpan* es necesario entender qué es *CPAN*.

De forma superficial, podemos entender CPAN, acrónimo de *Comprehensive Perl Archive Network*, como una red de módulos empaquetados con diversas funcionalidades. Al contrario que en este proyecto, los módulos están incluidos en un software escrito en *Perl,* que gestiona el tráfico de módulos que circulan por el mismo. *Meta::cpan* es una herramienta web desde la cual un usuario puede navegar por *CPAN* más cómodamente, evitando la consola.

2.2.5 *PyPI*[8]

Al igual que *meta::cpan*, *PyPI* es un sistema comparable al alcance de nuestro proyecto. En este caso se centra en el almacenamiento de paquetes escritos en *python.* Una diferencia que acerca *PyPI* a nuestro planteamiento es que, al contrario que *meta::cpan,* este sí que guarda los paquetes en un repositorio enorme.

Durante el estudio del mercado, se ha podido comprobar como plataformas parecidas a las dos comentadas anteriormente, *meta::cpan* y *PyPI*, son dos plataformas muy activas con miles de usuarios registrados y con decenas de miles de paquetes de código accesibles al público. Este dato nos muestra la utilidad que pueden ofrecer este tipo de sistemas, y la necesidad que generan. Podemos asegurar que, desde el primer día que el sistema se abra al público, recibirá un flujo constante de subidas y bajadas de módulos.

2.3 Herramientas de desarrollo

2.3.1 Recursos hardware

***Acer aspire E5-571G-51WG**:*

Soporte sobre el cual se ha desarrollado el proyecto.

2.3.2 Recursos software

Intellij

Ha sido el entorno de programación escogido para el desarrollo de la ampliación de la plataforma *Snap!* . El editor facilita la programación al usuario e incorpora la opción de sincronizar el código con *GitHub* mediante el controlador de versiones *Git*.

Git

Git es un controlador de versiones que ha estado presente durante todo el proceso de desarrollo. Como ya se ha comentado anteriormente, se ha encargado de supervisar las diferentes versiones de la nueva plataforma. El servidor responsable de la aplicación web exigía el uso del controlador para cargar las nuevas versiones. Finalmente ha sido y será el responsable de controlar el estado del repositorio central de módulos, localizado en el servidor *http*.

Kate

Ha sido el editor de texto escogido para el desarrollo de la aplicación web. Se ha escogido por su relación simplicidad/usabilidad.

Cloud9[9]

Es un entorno de desarrollo online y ha sido escogido para el desarrollo del servidor *http*. Su utilidad y relevancia pasa por dos de sus propiedades, interesantes a la par que necesarias. Por un lado, permite la edición de texto en la nube y revisar el código sintácticamente en tiempo real agilizando la programación. Por otro lado, nos permite ejecutar el script desde el mismo entorno, en otras palabras, nos permite mantener el servidor activo (durante un tiempo limitado).

Django[10]

Es un framework escrito en *python* orientado a la programación de aplicaciones web. Su objetivo es el de facilitar su desarrollo. La aplicación web se ha apoyado en este framework para agilizar su implementación.

Heroku[11]

Es una plataforma que nos ofrece el servicio de computación en la nube. Nos permite ejecutar nuestra aplicación web en la nube. De esta forma, la ejecución pasa a depender del servicio, cosa que nos permite desentendernos de su estado mientras no estamos conectados. Pese a las limitaciones que supone el hecho de tener una cuenta gratuita, los servicios que nos ofrece son suficientes para la puesta a punto de esta primera versión funcional de la aplicación web que se propone en este proyecto.

3 **Alcance del proyecto**

3.1 Objetivos

El objetivo global del proyecto es que los desarrolladores puedan exportar bloques, con una funcionalidad concreta y definida, a un repositorio central remoto. Todos los usuarios tendrán la capacidad de consulta y modificación de sus módulos.

Los objetivos como trabajo de final de grado descritos secuencialmente serían los siguientes:

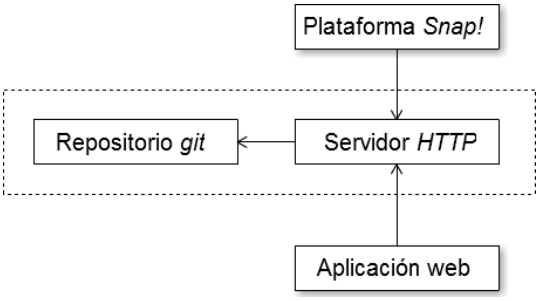
- Reunir ir valorar toda la información previa a la implementación, como el estado en el que se encuentra el terreno sobre que va a abordar el proyecto, su viabilidad aproximando una planificación temporal aproximada, su alcance etc.
- Considerar si disponemos de todos los recursos necesarios para su realización
- Implementar todos los componentes necesarios para la puesta a punto del repositorio
- Probar la versión final de la implementación para asegurarnos de que todas las funcionalidades que ofrece el sistema no tienen fallos
- Documentar el procedimiento de cada una de las fases por las que ha pasado el proyecto, concretamente en esta memoria.

Respecto a la parte más académica del trabajo, el proyecto debe abordar las siguientes competencias técnicas

- Demostrar el conocimiento de los fundamentos teóricos de los lenguajes de programación y las técnicas de procesamiento léxico, sintáctico y semántico asociados, y saber aplicarlos para la creación, el diseño y el procesamiento de lenguajes
- Definir, evaluar y seleccionar plataformas de desarrollo y producción hardware y software para el desarrollo de aplicaciones y servicios informáticos de diversa complejidad.
- Demostrar el conocimiento de los fundamentos, de los paradigmas y de las técnicas propias de los sistemas inteligentes, y analizar, diseñar y construir sistemas, servicios y aplicaciones informáticas que utilicen estas técnicas en cualquier ámbito de aplicación
- Desarrollar y evaluar sistemas interactivos y de presentación de información compleja y su aplicación en la resolución de problemas de diseño de interacción persona computador
- Implementar software de búsqueda de información (information retrieval)

3.2 Alcance del proyecto

Se ha optado por implementar todos aquellos elementos que pueda necesitar un usuario, para que el resultado no requiera de trabajo adicional y sea usable desde el primer momento. Los elementos seguirán la estructura que se plantea en la figura 1, y que consta de un servidor *http* que controle las peticiones hacia el repositorio y dos puntos de acceso totalmente independientes al mismo.



1. estructura propuesta en este proyecto

Servidor *http*

El servidor *http* propuesto en esta implementación está diseñado para maximizar la velocidad de acceso con el objetivo de maximizar el número de peticiones al servidor, que no serán más que lecturas y escrituras al repositorio remoto.

Se ha optado por utilizar el lenguaje *python* para su escritura porque ofrece numerosas bibliotecas orientadas a este tipo de proyectos, facilitando su desarrollo y detección de errores. La localización física del servidor se encuentra en una plataforma web que nos permite mantener el servidor en ejecución durante un largo periodo de tiempo. Otra ventaja del portal web es que nos permite editar los documentos directamente desde la interfaz, de forma que nos identificará los errores al instante si es que encuentra algún motivo por el cual aquel servidor no ha podido arrancar. La otra ventaja que tiene es el hecho de que no necesita que el portal web esté abierto para que el servidor esté activo (*Cloud9*).

Repositorio *git*

El servidor *http* alberga y gestiona un repositorio *git* cuyo objetivo es el de almacenar la información del módulo, que representaremos como un documento en formato *xml*. Dicho archivo guarda los datos referentes a la lista de bloques, su autor, una pequeña descripción etc.

Los módulos están reunidos por usuario de forma que dos usuarios puedan ser los autores de dos módulos con el mismo nombre, dando dos implementaciones válidas para una misma solución. En la práctica, un desarrollador no tiene ningún interés en generar bloques cuyo resultado sea parecido al que otro usuario haya publicado, a no ser que considere que el resultado es mejorable.

Actualización de *Snap!*

Tal vez la modificación de la plataforma *Snap!* para que pueda acceder directamente al repositorio, es el elemento más interesante de la estructura que se propone (y la más costosa temporalmente). Se ha dotado al sistema de una nueva

herramienta capaz de interactuar con una nube de bloques (código al fin y al cabo), importando contenido de otros desarrolladores directamente al entorno de trabajo. Asimismo, permite al usuario subir su código a esa nube. Desde ahora, se podrán desarrollar proyectos de forma más rápida porque contaremos con código complejo ya implementado por terceras personas. Los usuarios con poca experiencia también podrán sacar provecho de la ampliación porque tendrán acceso a código complejo que tal vez no sean capaces de implementar de forma efectiva.

Aplicación web

El elemento que culmina el sistema es la aplicación web que hemos desarrollado con el objetivo de poder acceder al repositorio de forma intuitiva siendo este capaz de permitirnos ver y descargar el contenido de todos los módulos que componen el repositorio. Como se explicará más adelante, el listado de módulos que se muestran, será el resultado de una simple búsqueda que nos ofrece la aplicación. La aplicación web (de la misma forma que como lo hace la interfaz de *Snap!* propuesta) pondrá a prueba la robustez de los elementos forzando la comunicación entre el servidor y el repositorio. La interfaz es muy sencilla pero incluye todas aquellas funcionalidades que un desarrollador de *Snap!* va a exigir a la aplicación.

Consideraciones

El repositorio será totalmente funcional en cuanto a los accesos se refiere y tanto la aplicación web como *Snap!* podrán comunicarse sin problemas con el mismo. No obstante, pese a que el resultado de este proyecto es totalmente funcional, tiene dos carencias que se han comentado superficialmente pero que consideramos que no entran en el alcance del proyecto. En cualquier caso el resultado es totalmente funcional.

La primera es tal vez la más importante. Se trata de que el servidor tiene un problema serio de seguridad. No comprueba la única condición que se ha de cumplir para que un usuario pueda modificar un módulo y es simplemente que ha de ser el autor y propietario del módulo. Al evitar la comprobación, cualquier usuario puede modificar cualquier elemento del repositorio. No se ha considerado en el alcance del proyecto porque, por un lado, requeriría un tiempo extra de implementación que probablemente

retrasaría la entrega del proyecto y no nos aseguraría que los responsables lo considerasen tan robusto como debiera, y por otra parte, consideramos que los responsables del filtro de seguridad que existe actualmente para gestionar los inicios de sesión, se podría exportar fácilmente.

La segunda es que la parte visual de la aplicación web (*html* y hojas de estilo) no está demasiado trabajada. A niveles prácticos no tiene ningún efecto negativo pero no atrae demasiado el interés del usuario. Toda la parte compleja del sistema junto a sus funcionalidades está implementada, por lo que no se ha considerado dedicar mucho tiempo a su diseño. Como que todas las funcionalidades de la aplicación web están cubiertas, un desarrollador web podría dedicarse al diseño sin saber *python*.

3.3 Posibles obstáculos

El principal obstáculo al que se ha de enfrentar cualquier proyecto en general y este en particular, es el de no poder seguir el ritmo de trabajo que se había fijado inicialmente en la planificación temporal. Un retraso puntual suele provocar retrasos significativos debido a que habrán partes del proyecto que no se podrán llevar a cabo porque otras de las que dependían, no están terminadas. Es por ello que hay que fijar una planificación temporal lo suficientemente flexible como para absorber dichos retrasos puntuales antes de que vayan a más. El haber paralelizado la implementación de las funcionalidades, nos minimiza el impacto que tendría el hecho de que se produjese este contratiempo porque el retraso no se acumularía a las siguientes iteraciones.

El otro principal obstáculo no depende ni de nuestra planificación, ni de nuestro diseño/implementación, ni de nuestro planteamiento, y por eso es el más difícil de superar, por no decir imposible. El hecho de no poder controlar una parte del proyecto, nos expone a errores desconocidos que no sabríamos resolver. Nos estamos refiriendo a las herramientas *Heroku* y *Cloud9*. Antes se ha comentado la necesidad (y dependencia) que tiene este proyecto de ambos sistemas por lo que no pueden fallar. En el caso de que sean inaccesibles, por un cambio en su política de servicios, o bien por un fallo técnico, no podríamos ni abrir el servidor ni ejecutar la aplicación. La única solución sería buscar otra plataforma que nos ofreciera los mismos servicios, pero las alternativas son muy escasas.

Un tercer obstáculo es que nuestro entorno de trabajo, el ordenador en este caso, sufriese un fallo catastrófico irremediable que lo dejara inutilizado. La consecuencia sería la pérdida de todo el trabajo realizado hasta el momento. Este no es peor que el obstáculo comentado anteriormente aunque lo parezca porque en este caso sí que tendríamos la capacidad de continuar con el proyecto, aunque fuese empezándolo desde cero. Evitarlo es muy sencillo. Crearemos una copia de seguridad en un repositorio controlado por *git* que alojaremos de forma remota en la página web *GitHub*. Al final de cada sesión de implementación o simplemente cuando lo consideremos, guardaremos en la nube nuestro trabajo como una nueva versión del código. Con esta técnica, siempre podremos recuperar versiones antiguas de nuestro proyecto.

3.4 Metodología

Para el desarrollo de este trabajo se ha optado por seguir un modelo ágil para asegurar su flexibilidad. El modelo que seguirá va a ser *Scrum* por el hecho de que permite desglosar el proyecto en secciones tan pequeñas como sea necesario facilitando su planificación temporal y su prueba de forma individual. Para sacar el máximo provecho de esta metodología es necesario hacer el despiece de forma inteligente buscando que las partes tengan un peso similar dentro del proyecto y que sean lo más paralelizables posibles. Como que se trata de una estructura con tres elementos que se han de diseñar e implementar de forma secuencial, no las trataremos de forma paralela. El orden se ha escogido ha sido el acceso desde Snap! en primer lugar, el servidor http + repositorio git en segundo lugar, y la aplicación web en último lugar.

Para cada uno de ellos, dividiremos su *backlog* en el diseño, implementación, testeo y, una vez terminado, testeo acoplándolo a los demás elementos. En el caso del acceso desde Snap!, será necesario un análisis exhaustivo del código desde el que partimos antes de proceder a su diseño. Este proyecto es especialmente paralelizable en funcionalidades. Por ejemplo, si estamos diseñando el modelo de consultas hacia el repositorio, podemos añadir elementos de forma manual sin haber implementado la funcionalidad de añadir. Del mismo modo, si estamos diseñando el modelo de publicación/modificación del repositorio, podemos ver los resultados de dicho cambio consultando su nuevo estado, sin necesidad de realizar una petición formal de tipo consulta.

3.5 Métodos de validación

El resultado se evaluará en dos fases. En la primera se probarán las características de cada elemento de la estructura por separado. En el caso de la aplicación web se probarán los enlaces entre las diferentes páginas, por ejemplo. En el caso de la ampliación de *Snap!* se probará la interacción entre las nuevas estructuras y las anteriores, la comunicación entre los elementos de una ventana, entre otros. Por último, en el caso del servidor *http*, se revisará el diseño de las *URI*s, se verificará que la respuesta a los distintos tipos de peticiones es la adecuada, entre otros. En la segunda se evaluará como un único sistema y no como varios elementos individuales. Consistirá en realizar peticiones de todo tipo y desde todos los puntos de acceso mientras se monitoria el flujo de datos que circulan por el servidor y el estado del repositorio después de cada una de ellas.

La validación más importante se ha llevado a cabo prácticamente cada semana en las reuniones de seguimiento con el director del proyecto. En ellas se ha supervisado el desarrollo del trabajo asegurándonos de que aquellas que validábamos eran robustas.

4 **Planificación temporal**

El desarrollo temporal del proyecto se aleja mucho de la planificación inicial que se planteó. El punto de inflexión que determinó que el proyecto debía retrasarse hasta finales de octubre ocurrió en el mes de abril donde, por razones personales, el proyecto quedó totalmente bloqueado.

4.1 Fases y descripción de las tareas

4.1.1 Planteamiento del TFG

El desarrolló el proyecto empieza con una primera reunión con el director y el ponente para enmarcar claramente el trabajo. Hay que tener muy claros los objetivos, el alcance, el sujeto del cual se parte etc., antes de definir las fases por las que deberá pasar el trabajo, en otras palabras, es necesario conocer el lugar de inicio y final antes de empezar a plantearse cuál va a ser el camino. Esta es la tarea más importante de todas

porque, si el punto de partida no es sólido o no está bien definido, el resultado final puede no coincidir con las expectativas que se querían alcanzar inicialmente.

4.1.2 Análisis del material, los recursos y las herramientas

Después de un primer planteamiento del proyecto, es necesario dominar, entender y familiarizarse con el material con el que se va a realizar la parte importante del trabajo que es la implementación. En cada una de las tareas incluiremos la implementación de cada una de sus funciones por pequeña que parezca para asegurarnos de que se ha hecho una buena lectura.

Familiarización con Morphic.js

Tal vez la parte más pesada: analizar y entender la herramienta con la que se va a trabajar una gran parte del trabajo que es el código de *Morphic.js*. Puede interpretarse como una tarea prescindible puesto que no hace falta conocerla al milímetro para empezar con la implementación ya que es bastante intuitiva. Sería un error empezar el trabajo con este planteamiento. Existe esa tentación porque la suma de los archivos base tienen una extensión total de más de 10.000 líneas de código. Para este análisis se dedicarán dos semanas para comprender y probar todas las estructuras de *Morphic.js*.

Familiarización con *Snap!*

Después es necesario comprender de qué forma *Snap!* hace uso de *Morphic.js*. Como ya conocemos ampliamente la forma que tiene *Morphic.js* de la tarea anterior, el análisis de la base de *Snap!* nos resultará fácil puesto que tan solo define una estructura global en la cual introduce objetos ordenados de forma jerárquica según le conviene. Es fácil seguir su estructura pese a que, en conjunto, tenga más de 15.000 líneas de código porque define los objetos uno a uno con todas sus características y funciones asociadas y, llegados a este punto, ya estamos familiarizados con ese proceso porque está definido en *Morphic.js*.

Pese a que es fácil interpretar su código, tenemos que prestar especial atención a una sección crucial para este proyecto que no parte de *Morphic.js*, la parte del *Cloud* de *Snap!.* Es necesario fijarse en la forma en la que *Snap!* se comunica con su *Cloud*.

Familiarización con la estructura 'física' de los bloques

El siguiente paso es entender la forma con la que *Snap!* exporta bloques localmente, es decir, de qué forma escribe una serie de bloques en un archivo plano en formato *xml* ya que éste va a ser el archivo que esta ampliación va a tener que guardar en el repositorio central.

Estudio de la plataforma *Cloud9*

Al ser un servició que desconocemos y que es ajeno a *Snap!,* es conveniente estudiar su entorno, para asegurarnos de que es una opción válida para mantener el servidor *http* con el repositorio *git*.

Estudio del servicio *Heroku*

Como en el caso de la plataforma *Cloud9,* es conveniente asegurarnos de que este servicio de computación en la nube satisface todas condiciones que requiere nuestra aplicación web.

4.1.3 Módulo de Gestión de Proyectos

A través del módulo *GEP* ampliaremos la visión teórica del trabajo. Se concretarán detalles referentes a su desarrollo como el alcance el proyecto/objetivos, planificación temporal/económica etc. Al ser tan solo una proyección, es una fase que solo depende de la primera visión general del proyecto, es decir la primera fase, por lo tanto se hará en paralelo con la segunda fase.

4.1.4 Diseño del proyecto

El paso previo a la implementación es siempre el diseño del proyecto. En la reunión con el director y el ponente se plantea la estructura que ha de tener un proyecto de estas características. Las mismas vendrán sujetas a las funcionalidades que han de soportar de cara a mantener la integridad del repositorio y a controlar los puntos de acceso al mismo. Por ejemplo, hay que decidir qué objetivos deberían cubrir cada uno de los puntos de acceso.

Esta ha sido la primera tarea que se ha visto modificada mientras el proyecto estaba en fase de implementación debido al retraso. Inicialmente el proyecto no contemplaba la implementación ni del servidor, ni de la aplicación web. La decisión de aplazarlo abrió la posibilidad a ampliar el alcance del proyecto con ambas estructuras. Por lo que, una vez que la ampliación de la plataforma estuvo implementada, se acordó dicha ampliación porque entonces sí que sería temporalmente alcanzable. Recordemos que el diseño y la implementación de la plataforma es la parte central de este proyecto.

4.1.5 Implementación

La implementación, junto al diseño, es la parte temporalmente más costosa porque hemos de programar varios elementos con distintos lenguajes de programación, su coordinación ralentiza su implementación. Separamos la implementación de la estructura en varias secciones referentes a cada elemento de la misma. Empezaremos por la ampliación de la plataforma *Snap!*, porque es la más costosa temporalmente, le seguirá la implementación del servidor *http* + repositorio *git*, y finalmente la aplicación web. Entre elementos tendremos que perfilar la implementación para que se acoplen correctamente unos a otros, junto a una serie de pruebas preliminares.

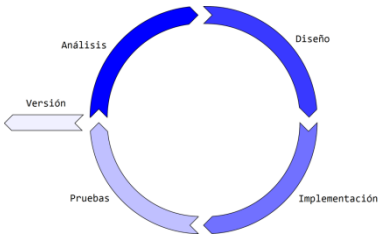
Como se ha comentado anteriormente, entre la implementación de la ampliación de la plataforma y del servidor, fue necesario el diseño de las demás estructuras porque fue entonces cuando se amplió el alcance de este proyecto.

4.1.6 Cierre del proyecto

En la última tarea se revisará la parte de implementación del proyecto y se realizarán las pruebas pertinentes para considerarlo como acabado. Una vez se ha comprobado que el resultado cumple con los objetivos planteados durante las reuniones entre el director, el ponente y el alumno, se procederá a su validación. Finalmente se documentará el proceso de desarrollo por el que ha pasado este proyecto junto a sus resultados, en este documento.

4.2 Iteraciones

4.2.1 Descripción de las iteraciones



Tal y como se ha planteado temporalmente el desarrollo del proyecto, la tarea dedicada a la implementación se divide en varias iteraciones de una longitud temporal similar, de una semana aproximadamente, que pasarán por las cuatro fases que se detallan a continuación.

Análisis

Se decide qué parte de código queremos implementar, comparando el estado de nuestro proyecto con la planificación inicial. La desviación que exista entre ambos estados determinará la cantidad de trabajo que debemos incluir en esta iteración.

Diseño

A pesar de que se ha dedicado gran parte del tiempo al diseño del proyecto antes de empezar con la implementación, es necesario describir la forma que queremos que tenga el resultado de aquella iteración. Esta fase se centra en el diseño a nivel de código mientras que el objetivo de la anterior es diseñar el sistema a nivel global.

Implementación

En esta fase procederemos a la implementación de aquella parte escogida y diseñada para esta iteración.

Pruebas

Dedicaremos la última fase a probar al nuevo código, tanto a nivel individual como a una incorporación al actual.

4.2.2 Lista de iteraciones

Ampliación de *Snap!*	
Explorar módulos	Implementación de la ventana*
	Implementación de las funcionalidades*
	Implementación de los accesos al repositorio
Publicar módulo	Implementación de la ventana*
	Implementación de las funcionalidades*
	Implementación de los accesos al repositorio
Servidor *http*	
Planteamiento	Diseño de la estructura del servidor y del repositorio
Peticiones	Implementación de las peticiones de tipo consulta
	Implementación de las peticiones de tipo publicación
	Implementación de las peticiones de tipo actualización y eliminación
Combinar	Adaptar las peticiones de *Snap!* al nuevo servidor
Aplicación web	
Planteamiento	Diseño de la aplicación y de las vistas como plantilla
Lista de módulos	Implementación de la vista con la lista de módulos acorde con la plantilla y sus pertinentes accesos al servidor
Lista de autores	Implementación de la vista con la lista de autores acorde con la plantilla y sus pertinentes accesos al servidor
Detalles de un módulo	Implementación de la vista con los detalles de un módulo acorde con la plantilla y sus pertinentes accesos al servidor
Detalles de un autor	Implementación de la vista con los detalles de un autor acorde con la plantilla y sus pertinentes accesos al servidor

*requieren 2 iteraciones

4.2.3 Diagrama de *Gantt*

Nombre	Duración	Inicio	Fin
Reunión con el director y el ponente	5d	01/02/2016	05/02/2016
Análisis del material, los recursos y las herramientas	10d	08/02/2016	19/02/2016
⊟Ampliación de Snap!	**50d**	**22/02/2016**	**29/04/2016**
⊟Exportar módulos	**25d**	**22/02/2016**	**25/03/2016**
Implementación de la ventana	10d	22/02/2016	04/03/2016
Implementación de las funcionalidades	10d	07/03/2016	18/03/2016
Implementación de los accesos al repositorio	5d	21/03/2016	25/03/2016
⊟Publicar módulo	**25d**	**28/03/2016**	**29/04/2016**
Implementación de la ventana	10d	28/03/2016	08/04/2016
Implementación de las funcionalidades	10d	11/04/2016	22/04/2016
Implementación de los accesos al repositorio	5d	25/04/2016	29/04/2016
Periodo de inactividad	25d	04/04/2016	06/05/2016
⊟Publicar módulo	**15d**	**09/05/2016**	**27/05/2016**
Implementación de las funcionalidades	10d	09/05/2016	20/05/2016
Implementación de los accesos al repositorio	5d	23/05/2016	27/05/2016
⊟Servidor HTTP	**40d**	**30/05/2016**	**22/07/2016**
⊟Planteamiento	**5d**	**30/05/2016**	**03/06/2016**
Diseño de la estructura del servidor y del repositorio	5d	30/05/2016	03/06/2016
⊟Peticiones	**25d**	**06/06/2016**	**08/07/2016**
Implementación de las peticiones de tipo consulta	5d	06/06/2016	10/06/2016
Implementación de las peticiones de tipo publicación	10d	13/06/2016	24/06/2016
Implementación de las peticiones de tipo actualización y e	10d	27/06/2016	08/07/2016
⊟Combinar	**10d**	**11/07/2016**	**22/07/2016**
Adaptar las peticiones de Snap! al nuevo servidor	10d	11/07/2016	22/07/2016
Periodo de descanso	15d	25/07/2016	12/08/2016
⊟Aplicación web	**40d**	**15/08/2016**	**07/10/2016**
Diseño de la aplicación, peticiones y plantillas	10d	15/08/2016	26/08/2016
Vista lista de módulos	5d	29/08/2016	02/09/2016
Vista lista de autores	5d	05/09/2016	09/09/2016
Vista detalles de un módulo	10d	12/09/2016	23/09/2016
Vista detalles de un autor	10d	26/09/2016	07/10/2016
Redacción de la memoria	25d	12/09/2016	14/10/2016
Preparación de la defensa	8d	17/10/2016	26/10/2016

2. lista de tareas planificadas junto a su duración prevista

Se ha incluido el diagrama completo en el anexo. Antes de pasar al análisis del diagrama, quería comentar que lamento la forma en la que está incluido en este documento. La única forma de que cupiese todo el diagrama sin perder información ha sido partiéndolo en cuatro secciones. Es tan extenso porque el periodo de tiempo que ha durado el desarrollo del proyecto también lo ha sido y, al estar dividido en diferentes tareas, no he visto conveniente el reducir el tamaño de las imágenes. Se ha complementado el diagrama con una representación más sintética de las tareas en la figura 2.

Vamos a empezar su análisis centrándonos en el periodo marcado de color rojo, durante el mes de abril y parte de mayo, donde el proyecto se detuvo por motivos personales. Hasta entonces se fue siguiendo, en la medida de lo posible, la planificación temporal definida inicialmente. Se aprecia que la actualización de la plataforma *Snap!* estaba prevista para mediados de mayo, sin embargo, se tuvo que detener cuando se empezaban a implementar las funcionalidades relacionadas con la publicación de módulos.

Ahora fijémonos al periodo posterior al periodo de inactividad. En este momento se tuvo que replantear una nueva planificación temporal que recuperase aquella parte del trabajo que debería estar implementada y que contemplase aquella ampliación del alcance del proyecto. Es por eso que vemos como la primera tarea que se deberá llevar a cabo una vez retomado el proyecto es la correspondiente a la implementación de las funcionalidades relacionadas con la publicación de módulos que había quedado pendiente. La planificación termina el día de la defensa del trabajo de final de grado, miércoles 26 de octubre.

Como es de suponer, este aumento significativo en la planificación temporal llevará irremediablemente a un aumento considerable del presupuesto, que se valorará a continuación.

5 **Gestión económica**

Tan importante es hacer una planificación temporal del TFG como estimar un presupuesto, margen de beneficios, amortizaciones de los recursos, etc. Para la estimación hay que tener siempre presente la planificación temporal ya que ambas están directamente relacionadas.

5.1 Introducción

Como en la planificación temporal, tendremos que comparar la previsión económica junto al presupuesto aproximado inicialmente, con aquella que se perfilo cuando se replanteó el proyecto una vez se supo que se prorrogaría hasta octubre.

5.2 Identificación de los costes

5.2.1 Costes directos

Recursos personales

El único recurso personal real que ha participado en el desarrollo de este trabajo final de grado ha sido el del estudiante por lo que el coste ha sido cero. En cualquier caso, aproximaremos los siguientes datos según el precio de mercado actual aproximado para valorar el coste que supondría desarrollar este proyecto.

En la siguiente tabla podemos ver los costes agrupados por roles en la **primera planificación temporal**, en el mes de febrero concretamente.

Rol	Horas estimadas	Remuneración en €/h	Coste estimado en €
Director de proyecto	150	20	3000
Analista	80	15	1200
Diseñador	50	10	500
Programador	100	15	1500
Responsable de pruebas	50	10	500
Tester	70	8	560
Coste estimado en €	500	-	7190

coste de cubrir gastos personales **inicial**

En la siguiente tabla podemos ver los costes agrupados por roles en la **segunda planificación temporal**, en el mes de mayo concretamente.

Rol	Horas estimadas	Remuneración en €/h	Coste estimado en €
Director de proyecto	225	20	4500
Analista	120	15	1800
Diseñador	125	10	1250
Programador	140	15	2100
Responsable de pruebas	60	10	600
Tester	80	8	640
Coste estimado en €	750	-	10890

coste de cubrir gastos personales **final**

Vemos claramente que el aumento en la parte del presupuesto asociado con los costes personales aumenta debido al aumento en horas de trabajo que le dedicará cada persona al proyecto. En la primera tabla observamos que estaban previstas 500 horas en un periodo de poco más de cinco meses con unas 5 horas al día aproximadamente, y en la segunda se han previsto unas 750 horas en un periodo de casi ocho meses aproximadamente (sin incluir el periodo de inactividad y de vacaciones). De ahora en adelante se considerará la segunda previsión económica.

En primer lugar, y como no podía ser de otra manera, la persona que va a estar más involucrada en el proyecto va a ser el director puesto que va a supervisar cada una de las fases para asegurarse de que el rumbo es el correcto. En segundo lugar, vemos como el analista, el diseñador y el programador le dedican una cantidad de tiempo similar. De la planificación temporal se puede deducir el por qué la cantidad de tiempo que le dedica el programador no es tan superior a los otros dos como es habitual. El diseñador tiene la responsabilidad de diseñar el proyecto a tres niveles: la combinación de los tres elementos a implementar, el diseño de los tres elementos por separado y el diseño de las peticiones hacia el repositorio. El analista deberá asegurarse de que el diseño propuesto es compatible con las herramientas que propone para llevarlo a cabo. Recordemos que cada elemento requiere de una herramienta específica que se debe analizar: *Snap!* depende de su extenso código fuente, el servidor requiere la plataforma *Cloud9* descrita anteriormente junto a bibliotecas de *python* que permiten interactuar con el repositorio *git* y la aplicación web requiere el servicio de computación en la nube *Heroku* también descrito anteriormente. Finalmente, el programador deberá implementar la compleja estructura propuesta por el diseñador. En último lugar, los encargados de realizar las pruebas pertinentes sobre el sistema y validar el resultado, se centrarán mayormente en hacer peticiones de todo tipo hacia el servidor monitorizando su flujo de datos y revisando constantemente la integridad del repositorio.

5.2.2 Costes indirectos

Como en el caso anterior, se considerarán los costes indirectos del respecto a la planificación propuesta en mayo.

Recursos hardware/software

```
Amortización   = (Precio*%de uso de su vida útil)/horas de uso
Coste estimado = (coste amortización*horas de uso)
```

Hardware						
Recurso	Precio	Horas de uso	uso de su vida útil	Vida útil	Coste amortización	Coste estimado
Acer aspire E5-571G-51WG	550	750	5%	5	0,036	27,5
Servidor *Cloud9*	0	500	0%	Inf	0	0
Servidor *Heroku*	0	250	0%	Inf	0	0
Servidor *Gantter*	0	10	0%	inf	0	0
Software						
Microsoft office	70	225	10	1	0,031	6,975
Intellij	0	250	0%	Inf	0	0
Kate	0	250	0%	Inf	0	0
Git	0	750	0%	Inf	0	0
Cloud9	0	500	0%	Inf	0	0
Gantter	0	10	0%	inf	0	0

Para el cálculo de la amortización se ha tenido en cuenta el precio del recurso, las horas que se va a usar durante el desarrollo del proyecto y el porcentaje que representan esas horas respecto al uso que se va a dar al recurso con temas no relacionados con el proyecto. La inversión en recursos es mínima ya que aquellos que son de pago van a utilizarse más allá del proyecto. A aquellos recursos gratuitos cuya vida útil desconocemos, les asignaremos una vida útil infinita lo que implica un uso de su vida útil del 0%. En el caso de los recursos hardware serán los servidores de los servicios que utilizaremos que daremos por hecho que siempre serán accesibles. En el caso de los recursos software serán aquellos programas que una vez instalados, no deberían dejar de funcionar nunca.

5.3 Coste de contingencia

Se consideró un margen de dos semanas para evitar que alguno de los riesgos a los que estaba expuesto produjese un retraso en la fecha de presentación del proyecto. En la planificación anterior no hemos tenido en cuenta este factor importante. Claramente el margen de dos semanas que se consideró fue sobrepasado ya que el retraso fue de un mes. Por ese motivo se tuvo que retrasar la fecha de entrega con todas las consecuencias económicas comentadas en esta sección. Lamentablemente no le podremos dejar un margen contra los retrasos porque la planificación temporal era muy ajustada.

El último gasto que añadiremos será una medida de contingencia del 10%:

Recurso	Coste estimado
Coste de recursos Personales	10890
Coste de recursos software/hardware	34,45
Coste parcial	10924,45
Coste contingencia	1092,445
Coste total	12016,895 ~ **12017**

5.4 Posibles desviaciones

La desviación más importante que puede sufrir el proyecto ocurriría durante la implementación del punto de acceso al repositorio desde *Snap!*, y es que el código base sufra un cambio inesperado, lo que supondría una revisión en profundidad de nuestro código para adaptarlo al nuevo estándar. Esta desviación está controlada porque se monitoriza constantemente el estado del código base de *Snap!* y se reservan periodos de prueba que aseguran la consistencia de nuestra implementación. La desviación produciría un retraso en el desarrollo ya que existe el riesgo de que parte de la base sobre la que se apoya la implementación hecha hasta el momento, difiriese considerablemente de la nueva.

Se produciría un aumento considerable en el presupuesto si el hardware sobre el que desarrollamos el proyecto, el ordenador, sufriese una avería y hubiese que cambiarlo. El remplazo no retrasaría el proyecto porque recordemos, guardamos una copia del proyecto en la nube.

6 **Sostenibilidad y compromiso social**

6.1 Impacto económico

Como ya se ha repetido varias veces, *Snap!* es un lenguaje desarrollado por la comunidad de desarrolladores y sin ánimo de lucro, poniendo el código al alcance del público desde el primer minuto. Los proyectos *opensource* suelen ser ambiciosos y motivan a que desarrolladores continúen con ese proyecto que, al fin y al cabo, también es para ellos. *Snap!* siempre será de código libre por lo que sólo se le podrá sacar un beneficio económico si se reimplementa con ese fin.

Bien es cierto que hay que considerar el coste económico que supone el mantenimiento de los servidores, en especial el de la aplicación web. En el desarrollo de este proyecto se han usado versiones gratuitas limitadas por lo que no ha supuesto un gasto añadido.

6.2 Impacto social

A nivel personal, el proyecto me ha ampliado el concepto que tenía de la programación visual *drag-and-drop* por bloques. Hasta el momento consideraba que esta forma de programar iba únicamente dirigida a programadores principiantes y que estaba muy limitada a nivel funcional como es el caso de *Scratch*. *Snap!* me ha demostrado que es posible la programación de proyectos complejos en diversas plataformas con un lenguaje de programación que, a primera vista, parece estar muy limitado.

Esta ampliación es realmente necesaria para conectar aún más a los usuarios de *Snap!* gracias a que ahora podrán compartir código directamente desde la interfaz en forma de bloques individuales. Hasta el momento la única forma de compartirlo era descargándolo de forma local y enviándolo por correo, lo que implicaba que los

desarrolladores debían estar en contacto. Desde ahora y aprovechando el concepto de la información en la nube, va a ser posible exportar ese código en la nube poniéndolo a disposición de la comunidad de desarrolladores lo que sin duda fomentará un aumento del uso de la plataforma.

6.3 Impacto ambiental

Es complicado estimar el impacto ambiental que supondrá la incorporación de la ampliación. Durante el desarrollo podemos afirmar que es prácticamente nulo puesto que no requiere elementos que puedan dejar huella, más allá del ordenador en el que se ha trabajado. Por otra parte, hemos de considerar la posibilidad de que el software se utilice en proyectos que sí que dejen una huella medioambiental ya sea porque traten de optimizar un sistema que sí perjudica al medioambiente reduciendo su contaminación, o porque participe en un proyecto que perjudica al medioambiente. En cualquier caso, estos datos no son aproximables.

6.4 Matriz de sostenibilidad

	PPP	Vida útil	Riesgos
Ambiental	8	16	0
Económico	8	9	-10
Social	7	15	-5
Rango de sostenibilidad	23	40	-15
	38		

Habiendo analizado el impacto que el proyecto provocará a nivel económico, social y ambiental, podemos cuantificar la sostenibilidad del proyecto en la matriz que hemos considerado. Resumiendo el análisis llevado a cabo y complementado con la matriz, considero que el proyecto es sostenible porque económicamente solo requiere el mantenimiento de los dos nuevos servicios, socialmente no provoca una huella ecológica significante y porque socialmente es un proyecto atractivo porque era una actualización muy esperada por la comunidad de desarrolladores de *Snap!*.

7 **Análisis de requisitos**

En esta sección se detallarán una serie de requisitos que ha de cumplir el servidor http, la aplicación web y la nueva interfaz de *Snap!* para validar el resultado. Los requisitos incluyen aquellos marcados por los objetivos de este elemento del proyecto, los exigidos por el cliente y las expectativas del usuario final.

7.1 Requisitos funcionales

Corresponden a aquellos requisitos más dirigidos al usuario final ya que hacen referencia a la usabilidad del producto desde un punto de vista más intuitivo que funcional. El usuario ha de ser capaz de realizar todas las funcionalidades que el cliente ha exigido de la forma más simple e intuitiva posible.

7.1.1 Servidor *http* y repositorio remoto

- Ha de poder acceder al contenido del repositorio y ofrecer su contenido en varios formatos según le convenga.
- Ha de poder modificar el repositorio por medio de peticiones
- Ha de poder identificar las peticiones inválidas más frecuentes y advertirle del motivo por el cual no ha podido procesarla

7.1.2 Aplicación web

- Ha de poder consultar la lista de módulos que contiene el repositorio junto a su descripción e información asociada
- Ha de poder filtrar la lista de módulos de forma que sólo aparezcan aquellos que contengan parte de un texto introducido
- Ha de poder consultar la lista de usuarios que han publicado un módulo como mínimo, junto a la lista de los módulos de los cuales él es el autor
- Ha de poder filtrar la lista de autores de forma que sólo aparezcan aquellos que contengan parte de un texto introducido
- Ha de poder acceder a la información de cada bloque de un módulo de forma individual

- Ha de poder descargar el contenido de un módulo
- Ha de poder consultar el contenido del módulo en formato *xml*

7.1.3 Acceso desde *Snap!*

Menú principal

- Ha de poder acceder a la opción de importar módulo
- Ha de poder acceder a la opción de publicar módulo

Importar

- Ha de tener acceso a la lista de módulos localizados en el repositorio remoto al abrir la ventana, sin necesidad de pedirlos.
- Ha de poder filtrar la lista de módulos de forma que:
    - Solo aparezcan aquellos que ha publicado (en el caso de que haya iniciado sesión)
    - Solo aparezcan aquellos que contengan parte de un texto introducido
- Ha de poder visualizar la lista de bloques que contiene el módulo seleccionado
- Ha de poder acceder a la información de cada bloque
- Ha de mostrar por pantalla la información y descripción del módulo seleccionado
- Ha de permitir descargar el conjunto de bloques que contiene el módulo seleccionado
- Ha de poder importar el módulo seleccionado al área de trabajo

Publicar módulos

Los requisitos que se exponen a continuación solo han de ser accesibles al usuario cuando este haya iniciado sesión, es decir, cuando el usuario se haya identificado.

- Ha de mostrar qué bloques se pueden publicar, es decir, qué bloques no ofrece la plataforma por defecto
- Ha de poder publicar un módulo con su nombre, descripción y lista de los bloques que desea que contenga

Actualizar/eliminar módulos

Los requisitos que se exponen a continuación solo han de ser accesibles al usuario cuando este haya iniciado sesión, es decir, cuando el usuario se haya identificado.

- Modificar el módulo seleccionado accediendo a la segunda ventana, siempre que sea propietario del módulo seleccionado
    - Conocer el contenido actual del módulo
    - Añadir / quitar bloques del módulo y modificar su descripción
    - Eliminar el módulo

7.2 Requisitos no funcionales

Corresponden a aquellos requisitos que ha de cumplir para que pueda soportar las funcionalidades expuestas en el apartado anterior. Por las características de este trabajo, el software va a ser el único elemento que va a tener que cumplir con los requisitos no funcionales

7.2.1 Servidor *http* y repositorio remoto

- Ha de ser capaz de responder a todo tipo de consultas, ya sean válidas o no
- Ha de mantener el repositorio actualizado y consistente después de cada petición
- Ha de ser capaz de responder de forma efectiva a aquellas peticiones válidas que no han podido llevar a cabo, por ejemplo, porque no ha encontrado el módulo/usuario requerido

7.2.2 Aplicación web

- Ha de ofrecer una visión atractiva de la lista de módulos/usuarios/bloques para su rápida consulta por parte del usuario
- Ha de poder interpretar el contenido (en formato *xml*) del módulo para extraer y ofrecer partes específicas del mismo
- Ha de poder reaccionar de forma adecuada a posibles errores que reciba por parte del servidor

7.2.3 Acceso desde *Snap!*

- Ha de seguir el estándar y las estructuras propuestas por los principales desarrolladores de *Snap!* en cuanto a código se refiere.
- Ha de resolver de forma efectiva y lo más discretamente posible los errores que envíe la respuesta del repositorio ya sea por un formato incorrecto o porque no se ha localizado el elemento requerido, entre otros.
- Ha de mostrar toda la información del módulo en la misma ventana de selección: nombre, descripción, información y el listado de sus bloques en forma de imágenes
- No ha de permitir a un usuario editar un módulo que no le pertenece
- No ha de permitir publicar un módulo a un usuario que no se haya identificado

8 **Diseño e implementación**

8.1 Servidor *http*

Es el eje transversal de todo el sistema que se ha diseñado, y por lo tanto es la pieza fundamental que decidía si el proyecto acababa en éxito o simplemente quedaba inacabado. Por él (y a él) pasarán todas las peticiones requeridas por ambas partes (*Snap!* y la aplicación web), por lo tanto no puede fallar cuando tenga que modificar la información del repositorio ni devolver más información de la que le pidan.

8.1.1 *Host*

Como prototipo, hemos optado por situar el servidor en el sitio web h*ttps://snaprepo-eledu.c9users.io*. Éste permite mantener un servidor en ejecución durante un periodo de tiempo aceptable, y nos permite editar el código directamente desde el sitio web facilitando su implementación. Los inconvenientes son que el servidor no puede estar activo indefinidamente y que no nos permite asignar una *URI* más apropiada. Insisto en que, como se trata de una primera versión, nos es suficiente para implementar el servidor de forma totalmente funcional. Estas limitaciones son debido a que utilizamos la versión gratuita del servicio. Una vez los responsables del proyecto global den el resultado por válido, se movería a un servidor oficial de *Snap!*.

8.1.2 Peticiones

GET

Como su nombre indica, un cliente realizará una petición de este tipo cuando necesite información sobre el contenido del repositorio, es decir, sobre la información de los módulos. Podemos dividir la complejidad de las peticiones de este tipo en dos grupos. Por un lado, el servidor ha de ser capaz de interpretar qué información le está pidiendo el cliente, ya sea referente a la estructura del servidor (relación módulo – autor), información general en forma de listas o bien al contenido de un módulo en particular. Por otro lado, hemos de proponer qué formato tendrán las *URI*s que recibirá el servidor teniendo presente que han de ser fácilmente interpretables por el servidor y, a ser posible, por un usuario.

La información que un cliente puede pedir es la siguiente:

- Una lista de todos los usuarios que han publicado un módulo como mínimo, es decir, una lista con aquellos usuarios que tienen algún módulo en el repositorio

URI	/users
Response	[estructura de **user**]
Data	-

- La información de un usuario a partir de su identificador (userName)

URI	/users/:*user_id*

Response	estructura de **user:**
	{
	‘name’: userName,
	‘modules’: [estructura de **module**]
	}

Data	-

- Una lista de todos los módulos que ha publicado un usuario

URI	/users/:*user_id*/modules
Response	[estructura de **module**]
Data	-

- La información sobre un módulo en concreto

URI	/users/:*user_id*/modules/:*module_id*

Response	estructura de **module**:
	{
	‘name’: moduleName,
	‘user’: userName,
	‘updated’: lastModificationDate,
	‘description’: moduleDescription
	}

Data	-

- La lista de módulos generados como resultado de aplicar la búsqueda de una subcadena en su nombre y descripción.

URI	/search?text=:*introduced_text*
Response	[estructura de **module**]
Data	-

POST

Éste tipo de peticiones le indicarán que un usuario quiere publicar un nuevo módulo. El mecanismo que utiliza para tratarlas es similar al que utiliza para tratar las de tipo *GET*. Una petición de tipo *POST*, y las dos que se comentan a continuación, modificarán el repositorio, en este caso para añadir un módulo como un archivo *xml*. Para que el servidor resuelva estas peticiones, no basta con tratar de forma adecuada la *URI*, sino que además deberá extraer los datos que acompañan a dicha petición, describiendo el nuevo elemento. La estructura es la siguiente:

URI	/users/:*user_id*/modules
Response	Mensaje de respuesta

Data	{
	‘name’: moduleName,
	‘blocks’: blocksXML,
	‘description’: descripción
	}

Extraerá el nombre del usuario que quiere publicar el módulo de la *URI* y construirá el documento final en formato *xml* con la información que reciba del cliente. La respuesta a la petición será un comentario informando de que el módulo se ha creado satisfactoriamente. En caso contrario, éste nos informará del motivo por el cual ha fallado la publicación (el módulo ya existe, falta información, etc.).

PUT

Las peticiones de tipo *PUT* se tratan de forma muy similar a las de tipo *POST*. Éstas corresponden a la modificación de un módulo existente. Realmente podríamos suprimir el tratamiento de las peticiones *PUT* agrupándolas a las de tipo *POST*. El motivo por el cual se ha optado por no hacerlo pese a ser más laborioso, es que subjetivamente representan un concepto diferente, por lo que de esta forma la estructura del servidor es más coherente, pese a que implique la repetición de código.

URI	/users/:*user_id*/modules/:*module_id*
Response	Mensaje de respuesta

Data	{
	‘name’: moduleName,
	‘blocks’: blocksXML,
	‘description’: moduleDescription
	}

El servidor modificará el módulo :*module_id* del usuario :*user_id*. Reemplazará el módulo existente por la nueva definición del módulo. Devolverá un error en el caso de que, o bien el usuario :*user_id* no existe, o bien el usuario :*user_id* no tiene el módulo :*module_id*.

DELETE

Por último, el servidor nos ha de permitir eliminar módulos almacenados en el repositorio. Para ello procesará peticiones del tipo *DELETE* preparadas para ese cometido. Al cliente le basta con facilitar el nombre del módulo y el nombre de su propietario para que pueda localizar y eliminar aquel elemento que recordemos, se identifica por su nombre y el nombre de su autor. Ambos estarán implícitamente especificados en la *URI*. Las peticiones tienen la siguiente estructura:

URI	/users/:*user_id*/modules/:m*odule_id*
Response	Mensaje de respuesta
Data	-

Complementando las cuatro peticiones anteriores, se ha optado por añadir una quinta que evitará posibles conflictos de seguridad. Se trata de *OPTION* que simplemente le hará saber al cliente que el servidor está preparado para procesar esos cuatro tipos de peticiones.

8.1.3 Estructuras y herramientas generales:

Antes de resolver cualquier petición, una rutina debidamente implementada segmentará la *URI* en varios elementos y parseará cada uno de los parámetros incluidos en la misma. Se ha intentado mantener la misma estructura de la *URI* para todos los tipos de peticiones.

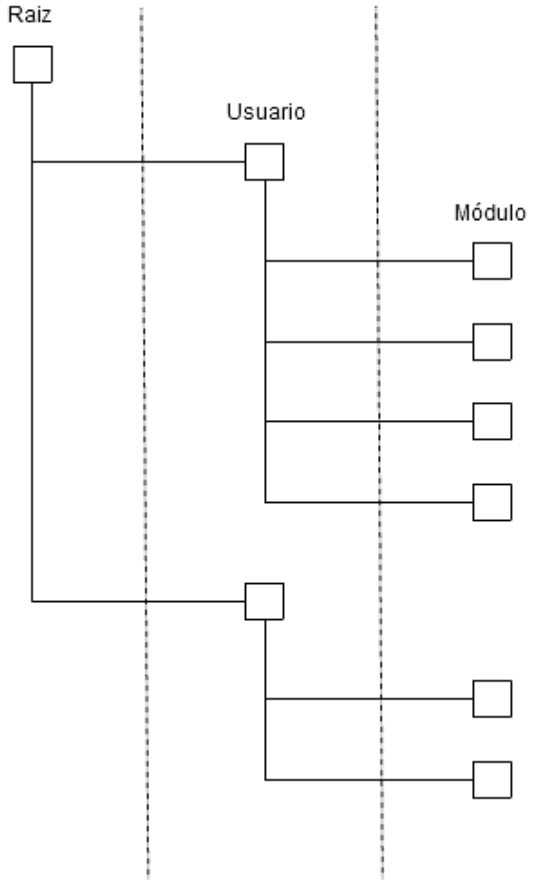
Cabe comentar un detalle muy importante respecto a las peticiones que se harán hacia el servidor, concretamente aquellas que implican una modificación del repositorio, es decir, las de tipo *POST*, *PUT* y *DELETE*. Es de suma importancia implementar algún tipo de control referente a quién puede modificar según qué módulo, puesto que un usuario dado sólo debería poder modificar aquellos módulos que ha publicado. La solución pasa por añadir algún tipo de filtro de seguridad que realice dichas comprobaciones. Sin embargo, se ha considerado no incluir dicha característica en el alcance del proyecto puesto que diseñar un sistema de seguridad invulnerable puede ser costoso en tiempo.

8.1.4 Repositorio *git*

El repositorio remoto *git* está almacenado físicamente junto al servidor *http* para agilizar la interacción entre ambos. Éste debe estar constantemente actualizado y ser consistente.

8.1.4.1 Estructura

Se ha optado por distribuir los módulos como un árbol de dos niveles. El primero no será más que una lista de carpetas con el nombre de los usuarios que han hecho alguna publicación. El segundo es una lista de archivos en formato *xml* con todos los módulos que contiene actualmente el repositorio. Éstos estarán agrupados por autor, repartiéndose en las carpetas creadas en el primer nivel. Podemos ver la representación gráfica de la estructura en la figura 3.



3. representación de la estructura del repositorio

Por lo tanto, el fichero del módulo :*module_name* perteneciente al usuario :*user_name* con la especificación del módulo es el /:*user_name*/:*module_name*.xml. Es útil situar el identificador del módulo en el nombre del documento porque de esta forma, no hará falta abrir cada uno de ellos para comprobar si se trata del que buscamos.

Se han utilizado dos bibliotecas específicas para el acceso y modificación del repositorio. Una facilita la navegación local entre los ficheros y la otra, centrada en *git*, permite interactuar con el repositorio, en otras palabras, aplicar comandos tan simples como *commit*s o preguntar por su estado. Las bibliotecas son *os* y *git respectivamente*.

8.1.4.2 Modificaciones

Cada petición procesada por el servidor a nivel de código debe aplicarse también al contenido del repositorio, en el caso de que sean peticiones del tipo *POST*, *PUT* o *DELETE*, comentadas anteriormente.

Añadir elemento (después de procesar una petición de tipo *POST*)

1. Navega por el repositorio en busca del directorio que contiene las publicaciones del usuario. Si no existe porque no ha hecho ninguna publicación, lo crea. Si existe, comprobará que no contenga un módulo con el mismo nombre.
2. Incorpora el nuevo módulo en el directorio.
3. Actualiza la versión del sistema

Modificar elemento (después de procesar una petición de tipo *PUT*)

1. Navega por el repositorio en busca del directorio que contiene las publicaciones del usuario. Comprueba que exista el usuario y que el usuario haya publicado un módulo con ese nombre.
2. Modifica el módulo especificado
3. Actualiza la versión del sistema

Quitar elemento (después de procesar una petición de tipo *DELETE*)

1. Navega por el repositorio en busca del directorio que contiene las publicaciones del usuario. Comprueba que exista el usuario y que el usuario haya publicado un módulo con ese nombre.
2. Elimina el módulo especificado
3. Si el usuario no tiene más módulos, borra ese directorio vacío.
4. Actualiza la versión del sistema

Es importante actualizar la versión del repositorio constantemente para mantener la consistencia del sistema.

8.2 **Aplicación web**

El objetivo de añadir una aplicación web a la estructura global del repositorio es ofrecer un segundo punto de acceso al contenido más fácil, rápido y visual. A través de ella accedemos a una visión más manejable del repositorio permitiéndonos agrupar módulos en listas y por usuarios, o permitiéndonos aplicar filtros de búsqueda más complejos. Al ser una herramienta de consulta, no nos permitirá realizar ninguna modificación del repositorio.

8.2.1 El sistema

Se ha desarrollado íntegramente en *python* a excepción de la sección de vistas que ha sido desarrollado usando *html* + hojas de estilo y *javascript*. La implementación se ha aprovechado el *framework Django*, escrito en *python*, especialmente dirigido a este tipo de proyectos web. Permite un fácil manejo del diseño vista-controlador por medio de plantillas *http* que deberán renderizarse apropiadamente antes de dar paso al intérprete de *http*, una vez el controlador haya resuelto la petición que ha provocado la acción. También ofrece una flexibilidad necesaria en el redireccionamiento de las direcciones así como parsear la *URI* y obtener parámetros localizados en la misma, por medio de coincidencias con expresiones regulares, de forma eficaz.

8.2.1.1 *Heroku*

Era necesario encontrar una plataforma donde situar la aplicación web físicamente y que nos permitiese ejecutarla durante un largo periodo de tiempo. Al ser un proyecto a medio plazo, era recomendable que esa plataforma nos ofreciese ese servicio de forma gratuita (con una serie de desventajas asumibles) para poder comprobar su correcto funcionamiento sin una inversión inicial. *Heroku* ha sido aquella plataforma que cumplía con nuestros requisitos y por lo tanto donde hemos situado esta primera versión de la aplicación. El soporte *Django* se acopla perfectamente a la plataforma minimizando la dificultad de su despliegue en la red.

La herramienta de sincronización entre la aplicación local y la aplicación en la red mediante el controlador de versiones *git*, es uno de sus puntos más fuertes. Éste permite subir la última versión de nuestro código utilizando el servidor de *Heroku* como un repositorio *git* remoto. De esta forma, tenemos el control del estado de nuestra aplicación y podemos gestionarla con los comandos tradicionales de *git*. Una vez el código se ha subido correctamente, la plataforma comprueba si mantiene la estructura necesaria para su ejecución y la despliega de forma automática. El resultado es que nuestra aplicación está iniciada en la dirección facilitada por *Heroku* y que nuestro repositorio local con el código está almacenado y sincronizado con el situado de forma remota dentro de la plataforma. Hay que remarcar que podemos hacer uso de todo el potencial que un controlador de versiones como *git* nos ofrece hacia el repositorio remoto.

8.2.1.2 *URI*s y vistas

En nuestro caso, *Heroku* nos ha facilitado una dirección web hacia nuestra aplicación que no podemos modificar porque no entra dentro de las funcionalidades que nos ofrece su uso gratuito. Una vez asumido, decidiremos la forma que queremos que tengan las *URI*s que determinarán a su vez la estructura de vistas que ofrecerá al usuario. La dirección que nos han ofrecido, y que corresponderá a la página principal de nuestra aplicación, es:

http://snapwebapp.herokuapp.com

El sistema ofrecerá la información sobre el contenido del repositorio por medio de las siguientes vistas:

Página principal

url: ../

Es la vista principal que muestra la aplicación una vez accedemos. Muestra el buscador de módulos/autores estándar como elemento principal de la vista.

Lista de módulos

url: */modules?text=:inserted_text*

Muestra el listado de módulos con el nombre, el autor, la descripción y fecha de la última modificación, como resultado de buscar en el nombre y la descripción, el texto introducido en el campo de texto del buscador estándar situado en la parte superior de la vista.

Lista de autores

url: */authors?text=:inserted_text*

Muestra el listado de autores acompañados de todos los módulos que han publicado, como resultado de buscar en el nombre de usuario, el texto introducido en el campo de texto del buscador estándar situado en la parte superior de la vista.

Información de un autor:

url: */authors/:author*

Muestra un listado con los módulos que ha publicado el autor en cuestión. A diferencia de la vista anterior, ésta permite obtener y visualizar más información acerca de los módulos que ha publicado el autor.

Información de un módulo:

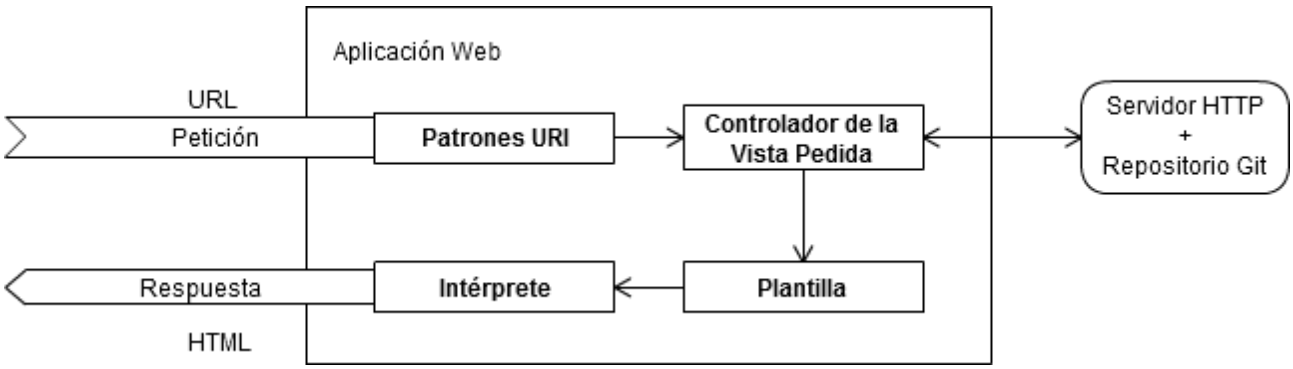
url: */author/:author_id/modules/:module_id*

Muestra de forma más detallada la información de un módulo en concreto. Por ejemplo, nos permite consultar los comentarios que ha introducido el autor en cada uno de los bloques que componen el módulo.

Además de la parte visual de la aplicación, ésta nos permite, a través de las mismas, realizar una serie de acciones sobre los módulos que completan la funcionalidad de la aplicación. Hay que resaltar tres de ellas en particular. En primer lugar, nos permite acceder al código que define el módulo. En segundo lugar, nos permite descargar el módulo como un documento de texto en formato *xml.* Por último, nos permite renderizar cada bloque individualmente facilitando su análisis. Se ha extraído el método de renderizado de la aplicación oficial de *Snap!* porque ya nos ofrece un método que ha sido probado y validado.

La aplicación se reserva una vista dedicada a informar de un error externo que pudiera surgir durante alguna de las peticiones hacia el servidor. El error más común es que el servidor se encuentre inaccesible ya que, como se ha comentado anteriormente, no nos asegura que esté activo de forma indefinida (debido a que estamos usándolo de forma gratuita).

El proceder de la ejecución a bajo nivel es el siguiente:

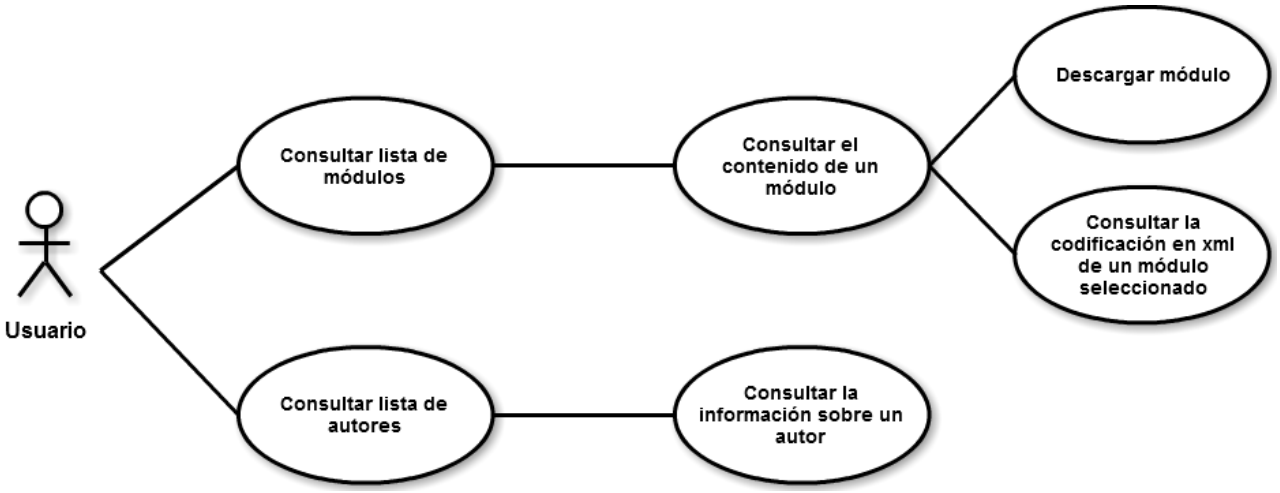


4. representación del proceso de tratamiento de una petición

El sistema identifica y vincula la *URI* requerida al controlador mediante expresiones regulares. Una vez identificado el controlador de la vista, procesará la petición junto a los parámetros de entrada que haya procesado y mandará renderizar la vista que considere más apropiada acompañada de los parámetros que necesite. Finalmente, un intérprete modificará la plantilla correspondiente a la vista acorde con los parámetros que han acompañado la petición de renderizado, y devolverá al navegador la plantilla traducida a *html*.

8.2.2 Casos de uso de la aplicación web

Las funcionalidades básicas que nos permite hacer la aplicación web se pueden resumir en el siguiente diagrama:



5. diagrama de casos de uso de la aplicación web

El diagrama, representado en la figura 5, se ha simplificado por una razón estructural. La aplicación web permite al usuario consultar la lista de módulos y usuarios desde cualquier vista de la aplicación puesto que se ha decidido que la herramienta de búsqueda siempre sea accesible. También hay que subrayar que desde la lista de autores y desde la información sobre un autor, es posible acceder a la información sobre alguno de sus módulos. Asimismo, desde la lista de módulos y desde la información sobre un módulo, es posible acceder a la información sobre su autor.

Caso de uso	Consultar lista de módulos
Actor	El usuario
Pre-condición	El servidor está operativo y la opción de mostrar módulos del buscador ha de estar seleccionada
Disparador	El usuario presiona el botón de buscar
Escenario principal	La aplicación muestra cualquier vista puesto que la opción de buscar está siempre visible
Extensiones	El usuario introduce texto en el área de texto del buscador. En ese caso, la aplicación filtrará la lista de módulos que muestra seleccionando aquellos que tengan ese texto o bien como parte del

	nombre o bien como parte de la descripción.
	El usuario selecciona la opción de mostrar módulo, y la aplicación dibuja los bloques que contiene el módulo tal y como lo haría la plataforma *Snap!*

Caso de uso	Consultar la lista de autores
Actor	El usuario
Pre-condición	El servidor está operativo y la opción de mostrar autores del buscador ha de estar seleccionada
Disparador	El usuario presiona el botón de buscar
Escenario principal	La aplicación muestra cualquier vista puesto que la opción de buscar está siempre visible
Extensiones	El usuario introduce texto en el área de texto del buscador. En ese caso, la aplicación filtrará la lista de usuarios que muestra seleccionando aquellos que tengan ese texto como parte del nombre
	El usuario selecciona la opción de mostrar módulo, y la aplicación dibuja los bloques que contiene el módulo tal y como lo haría la plataforma *Snap!*

Caso de uso	Consultar el contenido de un módulo
Actor	El usuario
Pre-condición	El servidor está operativo y ha encontrado la información sobre el módulo requerido
Disparador	El usuario selecciona un módulo desde la vista principal o desde la vista de su autor
Escenario principal	La aplicación muestra la información detallada del módulo centrándose en los bloques que lo componen
Extensiones	El usuario selecciona uno de los bloques. La aplicación dibuja su contenido tal y como lo haría la plataforma *Snap!*
	El usuario selecciona la opción de descargar módulo. La aplicación abre un panel de descarga que le permitirá descargar un archivo de texto con la traducción del módulo a formato xml. El nombre por

	defecto del archivo será el nombre del módulo
	El usuario selecciona la opción de ver el código fuente. La aplicación abre una nueva pestaña con la traducción del módulo a formato xml. El navegador interpretará el formato del texto facilitándonos su lectura

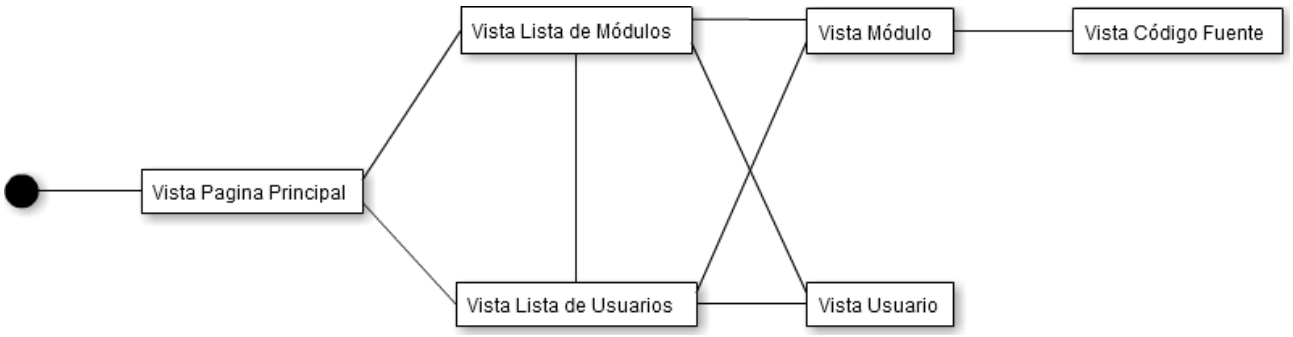
Caso de uso	Consultar la información sobre un autor
Actor	El usuario
Pre-condición	El servidor está operativo y el usuario requerido existe
Disparador	El usuario selecciona un usuario desde la vista principal
Escenario principal	La aplicación muestra la información básica sobre el usuario seleccionado y el listado de los módulos que le pertenecen
Extensiones	-

Caso de uso	Descargar módulo
Actor	El usuario
Pre-condición	El servidor está operativo y la vista con la información del módulo está activa.
Disparador	El usuario quiere descargar un módulo seleccionado
Escenario principal	1. El usuario selecciona la opción de 'Download' desde la vista que muestra la información del módulo 2. La aplicación prepara un documento en formato xml cuyo nombre es el nombre del módulo y abre una ventana de confirmación
Extensiones	-

Caso de uso	Consultar la codificación en xml de un módulo
Actor	El usuario
Pre-condición	El servidor está operativo y la vista con la información del módulo está activa.
Disparador	El usuario quiere consultar la codificación en xml de un módulo seleccionado
Escenario principal	1. El usuario selecciona la opción de 'Source' desde la vista que muestra la información del módulo

	2. La aplicación abre una nueva pestaña, indicando al navegador que se trata de código en xml y no en html, con el módulo interpretado en formato xml
Extensiones	-

8.2.3 Diagrama de vistas



6. diagrama de vistas de la aplicación web

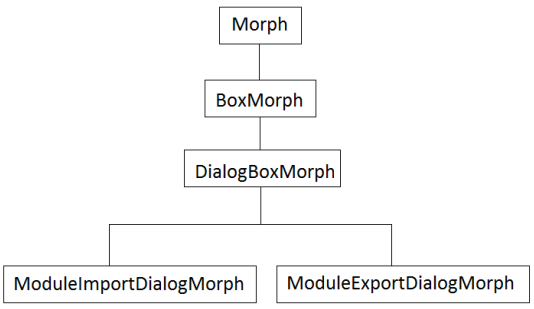
8.3 **Acceso desde la plataforma**

El último componente del sistema que compone la estructura propuesta en este proyecto es la modificación de la plataforma *Snap!* para dotarla de la capacidad de acceder al repositorio directamente desde su interfaz. De esta forma, el usuario no deberá descargar el módulo desde la aplicación web para después cargarlo en *Snap!* de forma manual sino que simplemente lo importará directamente. Para ello ha sido necesario descargar y modificar el extenso código (recordemos que ha sido implementado por numerosos desarrolladores) introduciendo aquellos elementos necesarios para lograr el objetivo propuesto.

8.3.1 Nuevas estructuras

Como ya se ha comentado anteriormente, *Snap!* está implementado íntegramente en *javascript*, para que pueda ser accesible desde el navegador, utilizando las herramientas que *Morphic.js* le ofrece. *Morphic.js* define un primer objeto al que llama *Morph* a partir del cual define los demás objetos extendiendo a dicho objeto. De la misma

forma, definirá más objetos que extenderán esa segunda serie que extiende a *Morph* y así sucesivamente. Lo que consigue es definir una estructura de objetos en forma de árbol a diferentes niveles en la que todos ellos tengan como raíz a *Morph* junto a sus propiedades, es decir, que todos compartan las características principales incluidas en él. Tal como está definido, el contexto permite una relación entre objetos de tipo *Morph* y, sabiendo que todos ellos parten de él, todos los objetos tendrán la capacidad de relacionarse y distribuirse en forma de árbol. Todos los objetos del entorno tendrán una dependencia con un objeto al que llamaremos 'padre' y con un grupo de objetos a los que llamaremos 'hijos', distribuidos en él. En nuestro caso definiremos dos objetos que extenderán al objeto *DialogBoxMorph* previamente definido: *ModuleImportDialogBoxMorph* y *ModuleExportDialogBoxMorph* siguiendo la estructura marcada en la figura 7.



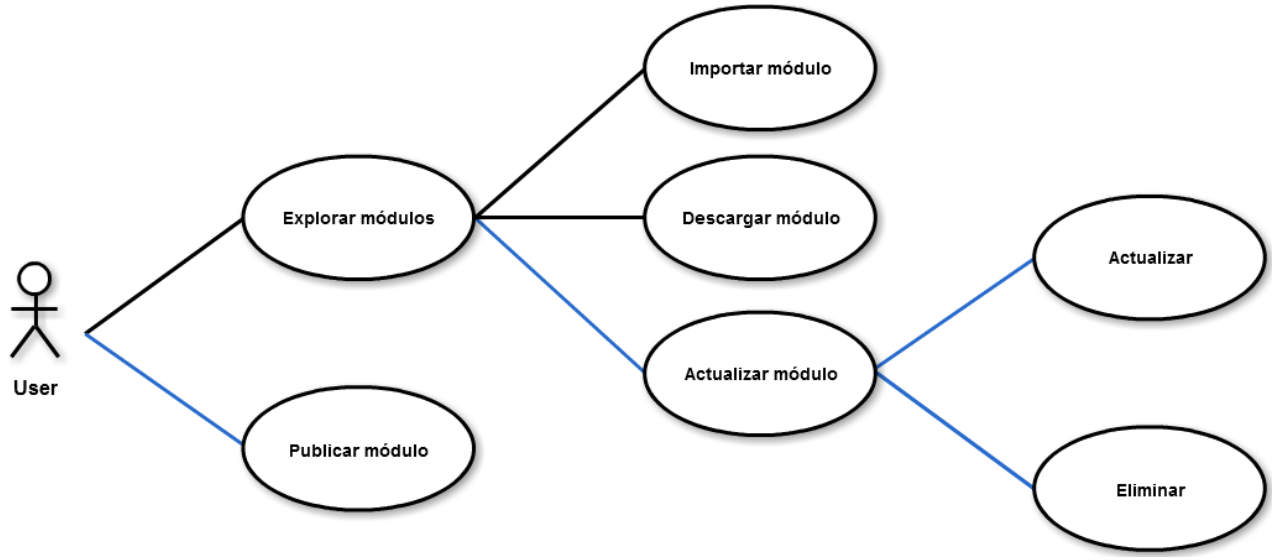
7. estructura jerárquica de los nuevos objetos

BoxMorph no es el único objeto que extiende a *Morph*, ni *DialogBoxMorph* es el único objeto que extiende a *BoxMorph* así como los dos objetos propuestos no son los únicos que extienden a *DialogBoxMorph*. Con este subárbol de la estructura jerárquica de nuestros objetos vemos cómo cualquier objeto de *Morphic* va a tener un *Morph* como raíz, heredando sus métodos y atributos. Por ello, al permitir una relación entre objetos *Morph,* es capaz de situar los elementos en el entorno, unos dentro de otros formando relaciones ‘padre’ -> ‘hijo’.

En el entorno de programación, ambos objetos serán 'hijos' directos de un *WordMorph* que alberga a todos los objetos que se visualizan en pantalla, un objeto al que llama *IDE_Morph*.

8.3.2 Casos de uso

A continuación se detallará la lista de casos de uso que ofrecerá el sistema. Como muestra la figura 8, podemos distinguir tres niveles de profundidad. El primero representa los casos de uso a los que el usuario tiene acceso desde el menú principal. El segundo representa los casos de uso a los tiene acceso desde la opción de importar módulo. El tercero representa los casos de uso referentes a la forma en la que el usuario quiere modificar su repositorio. Los enlaces pintados en color azul representan aquellas funcionalidades que solo son accesibles si hay una sesión iniciada.



8. diagrama de los nuevos casos de uso propuestos para Snap!

Las opciones de 'explorar módulos' (*Browse modules*) y publicar módulo (*Publish module*) accesibles desde el menú principal, se acoplan a la lista de las demás opciones predefinidas tal y como se puede apreciar en la figura 9.

9. nuevo menú de Snap!

Caso de uso	Explorar módulos
Precondición	No hay ninguna ventana abierta
Disparador	El usuario quiere ver la lista de módulos
Postcondición	El sistema ha abierto la ventana correspondiente y muestra una lista de módulos

Escenario principal	1. El usuario despliega el menú y selecciona la opción ‘Browse module’
Extensiones	1. El usuario introduce caracteres en el cuadro de texto y el sistema filtra los resultados por nombre y descripción
	1. Si el usuario se ha identificado, selecciona el *checkbox* con la etiqueta de *only my modules* y el sistema muestra únicamente aquellos que el usuario haya publicado
	1. El usuario selecciona un módulo y el sistema muestra los bloques que lo componen como una lista de imágenes correspondientes a cada bloque

Caso de uso	Publicar módulo
Precondición	No hay ninguna ventana abierta El usuario ha iniciado sesión
Disparador	El usuario quiere publicar un módulo
Postcondición	El sistema ha publicado los bloques que el usuario ha seleccionado, junto al nombre y la descripción que ha especificado.
Escenario principal	1. El usuario abre la ventana correspondiente a través del menú 2. El usuario especifica el nombre que identificará al nuevo módulo 2. El usuario selecciona los bloques que formarán el módulo 3. El usuario acopla una descripción al nuevo módulo
Extensiones	-

Caso de uso	Importar módulo
Precondición	La ventana correspondiente está abierta y hay un módulo seleccionado
Disparador	El usuario quiere importar un módulo
Postcondición	Los bloques del módulo seleccionado son ahora accesibles
Escenario principal	1. El usuario abre la ventana correspondiente a través del menú principal 2. El usuario selecciona el nombre de un módulo 3. El usuario presiona el botón ‘Import’
Extensiones	-

Caso de uso	Descargar módulo
Precondición	La ventana correspondiente está abierta y hay un módulo seleccionado
Disparador	El usuario quiere descargar un módulo
Postcondición	El sistema abre una ventana con la opción de descargar los bloques que contiene el módulo seleccionado
Escenario principal	1. El usuario abre la ventana correspondiente a través del menú perincipal 2. El usuario selecciona el nombre de un módulo 2. El usuario presiona el botón ‘Download’
Extensiones	-

Caso de uso	Actualizar módulo
Precondición	1. El usuario ha iniciado sesión
Disparador	El usuario quiere actualizar uno de sus módulos
Postcondición	El módulo actualizado contiene los bloques seleccionados con una nueva descripción especificada por el usuario
Escenario principal	1. El usuario accede a la ventana de explorar módulos 2. El usuario selecciona uno de sus módulos y presiona el botón de ‘Update’ 3. El usuario selecciona entre una lista qué bloques quiere que contenga el módulo 4. El usuario modifica, si lo ve conveniente, la descripción del módulo 5. El usuario presiona el botón ‘Update’
Extensiones	-

Caso de uso	Eliminar módulo
Precondición	1. El usuario ha iniciado sesión
Disparador	El usuario quiere eliminar el módulo
Postcondición	El usuario no tiene un módulo con ese nombre
Escenario principal	1. El usuario accede a la ventana de explorar módulos 2. El usuario selecciona uno de sus módulos y presiona el botón de

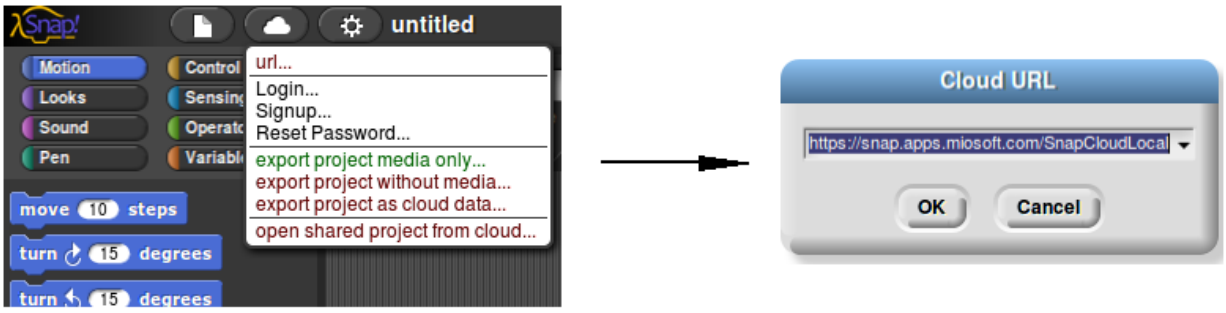
	'Update' 3. El usuario deselecciona todos los bloques 3. El sistema muestra un mensaje de confirmación alertando de que si no selecciona ningún bloque, el módulo será eliminado 4. El usuario confirma el mensaje de alerta
Extensiones	-

Login

Vamos a estar trabajando con el código de forma local, y por ello *Snap!* no nos va a permitir realizar según qué acceso a su servidor predefinido. Nos informará con la ventana emergente expuesta en la figura 10. El principal problema de este bloqueo es que no nos dejará iniciar sesión con ningún usuario, lo que implica que no podremos, a priori, hacer según qué funcionalidades que actuarían de una forma u otra en función de si hay una sesión iniciada o no. Por ejemplo, un usuario ha de haber iniciado sesión para poder actualizar o publicar módulos.

10 alerta de un error de conexión con el cloud de Snap!

Para ello es necesario cambiar la dirección contra la que vamos a realizar las peticiones. Simplemente debemos indicarle que estamos trabajando desde un servidor local y no desde el suyo. Accedemos a las opciones avanzadas desde la interfaz en modo desarrollador, seleccionamos la opción *url* y le indicamos que se conecte a *SnapCloudLocal*, tal y como muestra la figura 11.



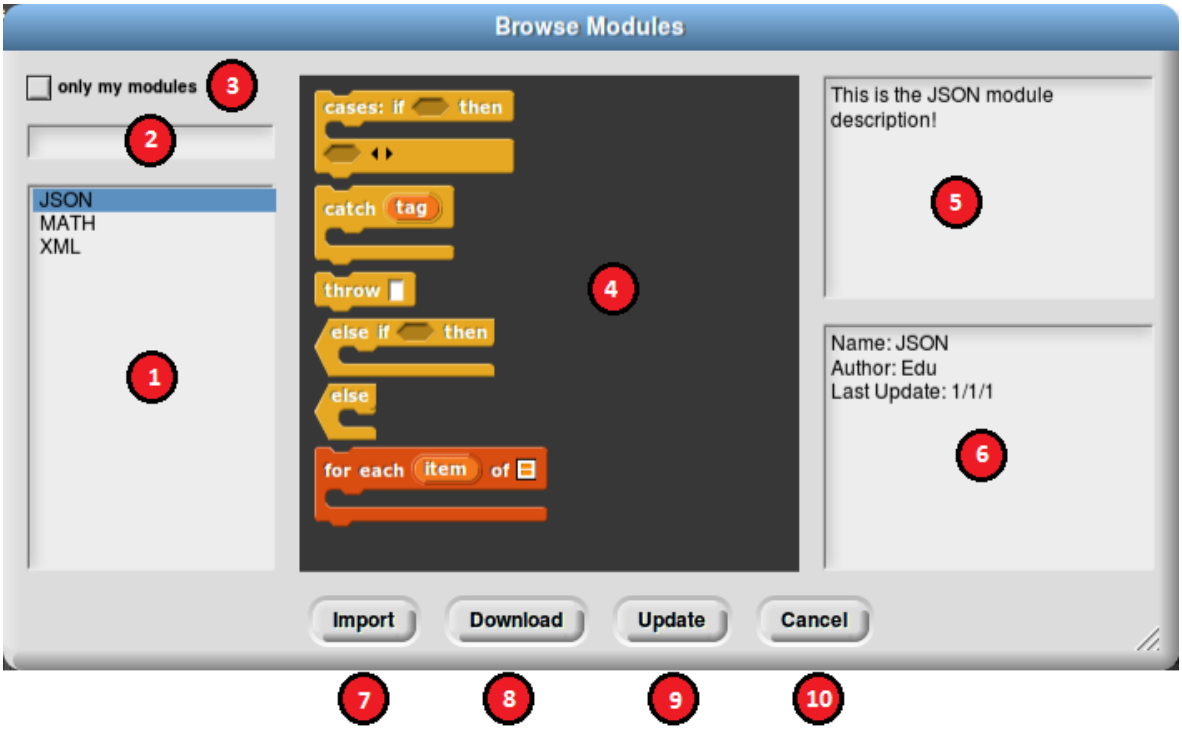
11 menú con las opciones de conexión con el cloud,
las opciones accesibles sólo en modo desarrollador se marcan en color

Otra forma de evitar la restricción es accediendo al entorno a través de un servidor que autorice el control de acceso a *Snap!* desde nuestro sistema. En nuestro caso hemos usado un simple servidor escrito en *python* para evitar complicaciones. No utilizaremos el servidor *http* que contiene el repositorio para evitar complicaciones.

Como se explica en la planificación temporal aproximada que hemos previsto para este proyecto, separamos la implementación de las funcionalidades en la parte de importación de módulos y la parte de exportación de módulos que corresponderán a dos opciones accesibles desde el menú principal. Cada una de ellas desplegará una ventana con una serie de elementos según la opción escogida. Un usuario que no haya iniciado sesión no podrá exportar módulos al repositorio por lo que la opción de publicar módulo quedará escondida si no se detecta ninguna sesión iniciada.

8.3.2.1 Explorar módulos, importar módulo y descargar módulo

En el caso de que seleccionemos la opción '*Browse modules*' del menú principal, veremos la lista de módulos que contiene el repositorio desplegando la siguiente ventana cuyas características están definidas en *ModuleImportDialogMorph* (figura 12).



12 El Morph tendrá 10 'hijos' de varios tipos, checkboxes, botones, listas, etc

Un objeto de tipo *ListMorph* [**1**] mostrará una lista con el nombre de todos los módulos que existen en el repositorio. Ésta va a ser la primera información que pedirá el *Morph* una vez ha recibido la orden de abrirse. Una entrada de texto de tipo *InputFieldMorph* [**2**] con la que podremos filtrar la lista de módulos de [**1**] para que nos muestre únicamente aquellos que tengan en el nombre parte del texto que introduzcamos. Un checkbox de tipo *ToggleMorph* [**3**] con el que podremos filtrar la lista de módulos para que nos muestre únicamente aquellos de los que somos propietarios. El *Morph* solo será visible si hay una sesión iniciada. Una lista de tipo *ScrollFrameMorph* [**4**] con una lista de imágenes de bloques. Una vez seleccionemos el módulo que queremos importar, hará una petición al repositorio con su nombre y éste nos devolverá el contenido del módulo con toda la información de cada uno de los bloques en formato *xml*. Una vez recogida, un método ya definido nos permitirá traducir ese documento a una lista de bloques. Una vez tenemos el módulo como una lista de bloques, desplegará la lista en forma de imágenes con la especificación de cada bloque. Si situamos el puntero sobre un bloque, veremos los comentarios que ha proporcionado el autor sobre aquel bloque. Junto a la lista de bloques, el documento *xml* contiene la descripción que ha proporcionado el autor del módulo, descripción que será visible en un objeto de tipo *ScrollFrameMorph* [**5**] y la

información en otro *ScrollFrameMorph* [**6**] parecido, con el nombre del módulo, el nombre del autor y la fecha de su última actualización. Finalmente los objetos [**7**], [**8**], [**9**], y [**10**] de tipo *PushButtonMorph* nos permitirán escoger el comportamiento final de la interacción que haremos con el bloque reclamado al repositorio. Después de cada función el *Morph* se autodestruirá.

- La opción [**7**], *Import*, añadirá a la lista de bloques por defecto aquellos incluidos en el módulo que hemos cargado, es decir, los que nos muestra el objeto [**4**].
- La opción [**8**], *Download*, nos dará la opción de descargar los bloques en formato *xml*. El archivo no será el mismo que encontramos en el repositorio, éste solo contendrá una serie de bloques.
- La opción [**9**], *Update*, sólo será visible si hay una sesión iniciada. Abrirá una ventana diferente con la opción de modificar el contenido del módulo que hemos seleccionado. El *Morph* no nos permitirá modificar el módulo si no somos el autor.
- La opción [**10**], *Cancel*, destruye el *Morph* sin realizar ninguna opción.

El primer acceso de *ModuleImportDialogMorph* que hará (de forma asíncrona) antes incluso que la propia construcción de la ventana, recibirá una lista con el nombre de los módulos que contiene el repositorio. Para ello hará una petición al repositorio a través de nuestro servidor con la *URI ~/search.* Al no incluir el parámetro *text* en la *query,* el servidor nos devolverá la lista completa de módulos. Lo haremos desde un nuevo método que hemos definido en *ModuleImportDialogMorph.* El método llamará a la función *getModules* que hemos añadido al objeto *SnapCloud*:

```
Cloud.prototype.getModules = function (callBack, errorCall, user) {
    var request = new XMLHttpRequest(),
    myself = this,
    url = user? "https://snaprepo-eledu.c9users.io/users/" + encodeURIComponent(user) + "/modules" :
"https://snaprepo-eledu.c9users.io/search";
    try {
        request.open(
            "GET",
            url,
            true
        );
        request.setRequestHeader(
            "Content-Type",
            "application/text"
        );
        request.onreadystatechange = function () {
            if (request.readyState === 4) {
                if (request.status === 200) {
```

```
                callBack.call(
                    null,
                    request.responseText
                );
            } else {
                errorCall.call(
                    null,
                    request.responseText,
                    'Error accessing the repository'
                );
            }
        }
    };
    request.send(null);
} catch (err) {
    errorCall.call(this, err.toString(), 'request error');
}
}
```

Éste devolverá, o bien la lista completa de módulos o bien la lista de módulos que pertenecen a un usuario en el caso de que el usuario lo haya definido como parámetro (es decir, cuando el parámetro no sea nulo). Una vez haya recibido la información, *ModuleImportDialogMorph* la almacenará como un *array* de objetos con la información de los bloques, tal y como se ha explicado anteriormente (el nombre del módulo, el nombre de su autor, la fecha de su última actualización y su descripción). En este momento la plataforma ya reúne toda la información necesaria para construir esa primera estructura.

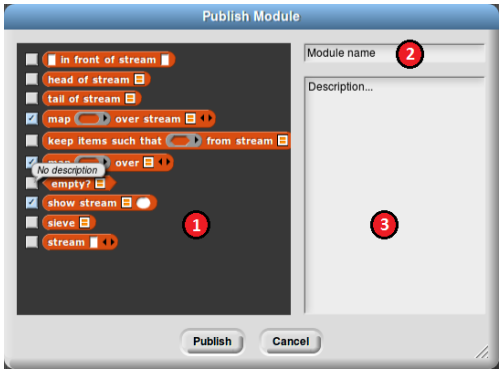
Para obtener el contenido de cada módulo que necesita el *Morph* para crear la lista de bloques, utilizaremos el método *getModuleContents* que hemos añadido al objeto *SnapCloud*. Para ello necesitaremos el nombre de usuario al cual pertenece el módulo y el nombre del módulo.

```
Cloud.prototype.getModuleContents = function (callBack, errorCall, user, module) {
    user = encodeURIComponent(user);
    module = encodeURIComponent(module);
    console.log(module)
    var request = new XMLHttpRequest(),
    myself = this;
    try {
        request.open(
            "GET",
            "https://snaprepo-eledu.c9users.io/users/" + user + "/modules/" + module + "/contents",
            true
        );
        request.setRequestHeader(
            "Content-Type",
            "application/text"
        );
        request.onreadystatechange = function () {
```

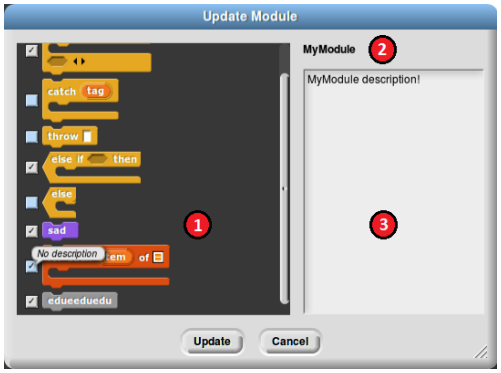
```
            if (request.readyState === 4) {
                if (request.status === 200) {
                    callBack.call(
                        null,
                        request.responseText
                    );
                } else {
                    errorCall.call(
                        null,
                        request.responseText,
                        'Error requesting the module contents'
                    );
                }
            }
        };
        request.send(null);
    } catch (err) {
        errorCall.call(this, err.toString(), 'Snap!Cloud');
    }
}
```

8.3.2.2 Publicar y actualizar

Las funcionalidades de *Publish* y *Update* despliegan ventanas prácticamente iguales (figuras 13 y 14) por lo que ha sido conveniente reunir ambas funcionalidades en el objeto *ModuleExportDialogMorph*. Un atributo al que llamaremos *task* y que sólo podrá tener los valores '*publish'* y '*update'*, identificará la funcionalidad que se está llevando a cabo. La ventana se desplegará si seleccionamos la opción de *Publish module* (*task* = '*publish'*) del menú principal o si seleccionamos la opción de *Update* desde el *Browse module* (*task = 'update'*). Recordemos que ambas funcionalidades no serán accesibles si no hay una sesión iniciada*:*



13 ventana correspondiente a *publish module*



14 ventana correspondiente a *update module*

Una lista de tipo *ScrollFrameMorph* [**1**] con una lista de *checkboxes*. Cada uno de ellos, como en el caso anterior son de tipo *ToggleMorph,* tendrán la imagen de un bloque

en vez de un texto. Mostrará aquellos bloques que no estén definidos por defecto, es decir, los importados o creados por el usuario. Como en *ModuleImportDialogMorph*, situando el puntero sobre un bloque veremos su descripción. Un objeto de tipo *ScrollFrameMorph* [**3**] con la descripción del bloque que queremos actualizar/publicar. Un objeto de tipo *TextMorph* [**2**] con el nombre del módulo que queremos actualizar/publicar.

Si queremos publicar un módulo:

Actuarán como parámetros de entrada de texto que darán nombre y descripción al nuevo módulo. Si no hemos seleccionado ningún bloque, el mismo objeto *ModuleExportDialogMorph* nos impedirá su publicación y nos dará una alerta con el motivo. Actuará de la misma manera si ya tenemos un módulo con el nombre introducido. Para hacerlo, si lo que intentaba era actualizar el módulo, tendrá que hacerlo vía la opción *Update* de *Browse modules.* Para acceder a la lista de módulos del usuario, hará la petición *getModules* de *Cloud*.

Si queremos actualizar un módulo:

Añadirá a la lista [**1**] los bloques que contiene el módulo que ha seleccionado mediante la petición *getModuleContents* de *Cloud* con el *checkbox* seleccionado. La entrada de texto [**2**] estará bloqueada con el nombre del bloque que estamos actualizando y en la entrada de texto [**3**] se cargará la descripción actual del módulo para ser modificada.

La modificación del repositorio se hará mediante otra petición *XMLHttpRequest* por el nuevo método que hemos definido como *getModuleContents:*

```
Cloud.prototype.exportModule = function (callBack, errorCall, contents, task) {
    contents['user'] = encodeURIComponent(contents['user']);
    contents['name'] = encodeURIComponent(contents['name']);
    var request = new XMLHttpRequest(),
        myself = this;
    try {
        if (task === 'update') {
            request.open(
                "PUT",
                "https://snaprepo-eledu.c9users.io/users/" + contents['user'] + "/modules/" +
contents['name'],
                true
```

```
            );
        } else {
            request.open(
                "POST",
                "https://snaprepo-eledu.c9users.io/users/" + contents['user'] + "/modules",
                true
            );
        }
        request.setRequestHeader(
            "Content-Type",
            "application/json"
        );
        request.onreadystatechange = function () {
            if (request.readyState === 4) {
                if (request.status === 200) {
                    callBack.call(
                        null,
                        request.responseText
                    );
                } else {
                    errorCall.call(
                        null,
                        request.responseText,
                        'Error exporting the module'
                    );
                }
            }
        };
        request.send(JSON.stringify(contents));
    } catch (err) {
        errorCall.call(this, err.toString(), 'Snap!Cloud');
    }
}
```

Si queremos eliminar un módulo:

El procedimiento es muy similar al que realizaríamos si quisiésemos actualizar un módulo. Una vez en la ventana dedicada a la actualización de módulos, deseleccionamos todos los bloques de la lista y le indicamos que queremos actualizar el módulo sin bloques. La plataforma recibirá una lista vacía de bloques y enviará un mensaje alertando de que como no ha seleccionado ningún módulo, éste será eliminado.

```
Cloud.prototype.deleteModule = function (callBack, errorCall, name) {
    var request = new XMLHttpRequest();
    try {
        request.open(
            "DELETE",
            "https://snaprepo-eledu.c9users.io/users/" + encodeURIComponent(this.username) +
"/modules/" + encodeURIComponent(name),
            true
        );
        request.setRequestHeader(
            "Content-Type",
```

```
            "application/json"
        );
        request.onreadystatechange = function () {
            if (request.readyState === 4) {
                if (request.status === 200) {
                    callBack.call(
                        null,
                        request.responseText
                    );
                } else {
                    errorCall.call(
                        null,
                        request.responseText,
                        'Error while deleting the module'
                    );
                }
            }
        };
        request.send(null);
    } catch (err) {
        errorCall.call(this, err.toString(), 'Snap!Cloud');
    }
```

9 **Conclusiones**

9.1 Introducción

Llegados a este punto del documento, se realizará un análisis del proceso de desarrollo que se ha llevado acabo junto a una comparación de los objetivos que se habían planteado desde un principio, y su resultado.

Se analizará la forma en la que se han resuelto los objetivos principales definidos en el alcance del proyecto, se valorará el futuro que tendrá el resultado final que sin duda es prometedor porque, como ya se ha dicho en varias ocasiones, formará parte de un sistema utilizado por innumerables desarrolladores y finalmente se realizará una valoración global del proyecto a nivel personal, centrado en el proceso de desarrollo y en cómo se han abordado las dificultades que han surgido en el mismo.

1.2 Resolución de objetivos

El análisis del resultado final del proyecto pasa por reunir los objetivos planteados y explicados anteriormente junto al alcance del mismo, y revisar si estos se han cubierto de forma adecuada.

La unión del conjunto de los elementos que componen el proyecto ofrecen al usuario la capacidad de almacenar/extraer código, en forma de módulos, de un repositorio central en la nube. Por una parte, el usuario puede interactuar con el repositorio exportando e importando módulos subidos por la comunidad de desarrolladores de *Snap!* directamente desde la plataforma web, ofreciendo una interacción directa entre el contenido importado y su entorno de trabajo. Por otra parte, el usuario puede utilizar la aplicación web construida como parte del proyecto, como otro punto de acceso al repositorio. La diferencia es que éste ofrece una visión más gráfica y atractiva del contenido del repositorio para una navegación orientada a la consulta de su contenido. Con esta valoración muy superficial del resultado, podemos afirmar que el proyecto ha cumplido los objetivos planteados en el alcance del proyecto. Bien es cierto que no podremos validarlo hasta revisarlo de forma exhaustiva elemento a elemento.

Sobre el servidor *http* y el repositorio remoto, no nos conformábamos sólo con que fuese accesible para la nueva ampliación de la plataforma y para la aplicación web que se propone, sino que también debía ofrecer un acceso intuitivo para que otras plataformas pudieran acceder fácilmente, es decir, los accesos tenían que seguir algún tipo de estándar. Para abordar el objetivo, se definieron de la forma más general posible, las estructuras que debían de tener las peticiones al mismo. Se le preparó para poder procesar todo tipo de peticiones y se trató de dar un significado semántico a las *URI*s.

Los objetivos relacionados con la aplicación web eran básicamente que el usuario pudiese consultar el módulo de la forma más cómoda posible. Ésta permite una exploración filtrada por módulos y autores, junto a un filtrado por texto opcional. Ofrece una visión atractiva sobre el contenido de un módulo, y la información sobre un usuario. También ofrece la lista de bloques que contiene un módulo prácticamente desde cualquier vista, permitiendo un primer reconocimiento de su contenido. También ofrece la posibilidad de descargar un módulo codificado en formato *xml* para su posterior

importación desde la plataforma, y la posibilidad de ver su codificación directamente desde el navegador. Los bloques se exponen en forma de imagen tal y como lo haría el entorno de *Snap!*.

Los objetivos que requerían una resolución lo más precisa posible, eran los de la última estructura, la ampliación de la plataforma *Snap!*. El por qué reside en que ésta es la que va a centrar todas las miradas de cara a un futuro cercano puesto que pasará a formar parte del día a día de los desarrolladores porque recordemos, aspiramos a que sea utilizada desde el primer minuto puesto que es una actualización muy esperada por la comunidad debido a su enorme utilidad. Por lo tanto, la robustez del sistema debía ser inquebrantable, es por eso que esta parte ha sido la que ha llevado más trabajo. Los objetivos se resumían en que el usuario tenía que poder acceder al repositorio para poder realizar tareas como importar, exportar, actualizar, borrar o descargar módulos. No solo consideramos que se han cumplido dichos objetivos sino que la solución propuesta en este proyecto es lo suficientemente robusta como para ser utilizada desde el primer momento (una vez se hayan resuelto los problemas comentados anteriormente).

La conclusión sobre el resultado final del proyecto una vez los objetivos se han revisado de forma exhaustiva, es que se puede considerar que ha cumplido con todas las expectativas. Llegados a este punto, hay que insistir en que el servidor tiene un defecto que no se ha solucionado porque no entraba en el alcance del proyecto. El defecto es básicamente que el servidor es vulnerable a modificaciones no autorizadas por el autor, es decir, le falta un filtro potente de seguridad para evitar modificaciones indeseadas. No se ha considerado incluir la resolución del problema en el conjunto de objetivos porque es moderadamente complejo y no se puede hacer una aproximación temporal de su resolución.

1.3 Futuro del proyecto

Como cualquier proyecto de estas características, sus expectativas de futuro pasan por el impacto que suponga la incorporación de la ampliación en referencia al uso de los usuarios que trabajan actualmente con el sistema.

Los clientes, en este caso desarrolladores fieles a este tipo de lenguajes, esperaban una actualización de este tipo. Últimamente, se está poniendo especial interés en utilizar internet como una herramienta de conexión entre personas más que como una herramienta de conexión entre documentos web. Ésta era una actualización muy esperada por la comunidad ya que ofrece un puente, prácticamente anónimo, entre desarrolladores. En estos momentos, la única forma de acceder al código de otros desarrolladores, era compartiendo proyectos enteros (a no ser que ya estuviesen en contacto). A partir de ahora, será posible acceder a unidades de código que previamente haya publicado algún desarrollador, acelerando la programación en este lenguaje.

El futuro inmediato del proyecto pasa por la aceptación y acoplamiento del mismo al código oficial (recordemos que es una plataforma *opensource*). Si este no es aceptado, se tendrían que modificar aquellas partes que entran en conflicto con la estructura actual.

1.4 Valoración personal

Ha sido satisfactorio participar en un proyecto *opensource* de estas dimensiones en el que están implicados tantos desarrolladores que trabajan día a día para mejorarlo. Acoplando la nueva funcionalidad a *Snap!,* estoy introduciendo una herramienta que unirá aún más a esa comunidad en expansión, no solo facilitando y acelerando su programación, sino también invitando a nuevos usuarios a probar dicha plataforma.

La comparación entre la planificación temporal y el tiempo real que ha durado el desarrollo del proyecto (el hecho de que se haya tenido que prorrogar), es una demostración de lo impredecible que era medir sus dimensiones. Tenía que desarrollar varios elementos dependientes y coordinados entre sí. Se valoró si era posible su finalización en el tiempo previsto. El resultado fue que el alcance del proyecto cubriría, en un principio, sólo el acceso desde la plataforma utilizando un repositorio remoto ya implementado ya que era la parte temporalmente más costosa y tenía que ser la más robusta. En cualquier caso, siempre se tenía presente que había un objetivo secundario que era la implementación de las otras dos estructuras. El retraso inesperado de un mes, forzó la prórroga de la entrega y fue entonces cuando se decidió abarcar también esos objetivos secundarios.

Uno de los retos a los que me he tenido que enfrentar para la culminación del proyecto ha sido que me he adentrado en un terreno que no dominaba con dos lenguajes que no dominaba. De hecho, en el caso del desarrollo del acceso desde la plataforma de *Snap!*, no solo trabajaba con un lenguaje que no controlaba, sino que además trabajaba sobre un código muy extenso que no podía adaptar. Sin embargo, a la pregunta de si lo volvería a hacer, respondo sin ninguna duda que sí. El enorme trabajo que me ha supuesto su desarrollo, me ha capacitado para enfrentarme a futuros proyectos de este tipo que hasta el momento no veía claros. Me refiero a la conexión entre los cuatro elementos y a los dos lenguajes en los que he profundizado.

Hemos preferido hacer llegar el resultado de este proyecto a los responsables de la supervisión de *Snap!* después de haberlo presentado como TFG. La satisfacción que recibiría a nivel personal si aceptan acoplar mi herramienta (sin demasiados cambios) sería indescriptible. Entiendo que tardaría un tiempo puesto que tendrán que mover el servidor, tal vez mejorar gráficamente la aplicación web y probablemente ordenar parte del código.

10 **Bibliografía**

[1] Página principal de *Snap!* - *http://www.snap.berkeley.edu/*

[2] Código fuente de *Snap!* - *https://www.github.com/jmoenig/Snap--Build-Your-Own-Blocks*

[3] Página principal de *Scratch* - *https://www.scratch.mit.edu/*

[4] Página principal de *Beetleblocks* - *http://www.beetleblocks.com/*

[5] Página principal de *Snap4Arduino* - *http://www.s4a.cat/snap/*

[6] Página principal de *App Inventor* - *http://appinventor.mit.edu*

[7] Página principal de *meta::cpan* - *https://metacpan.org/*

[8] Página principal de *PyPi* - *https://pypi.python.org/pypi*

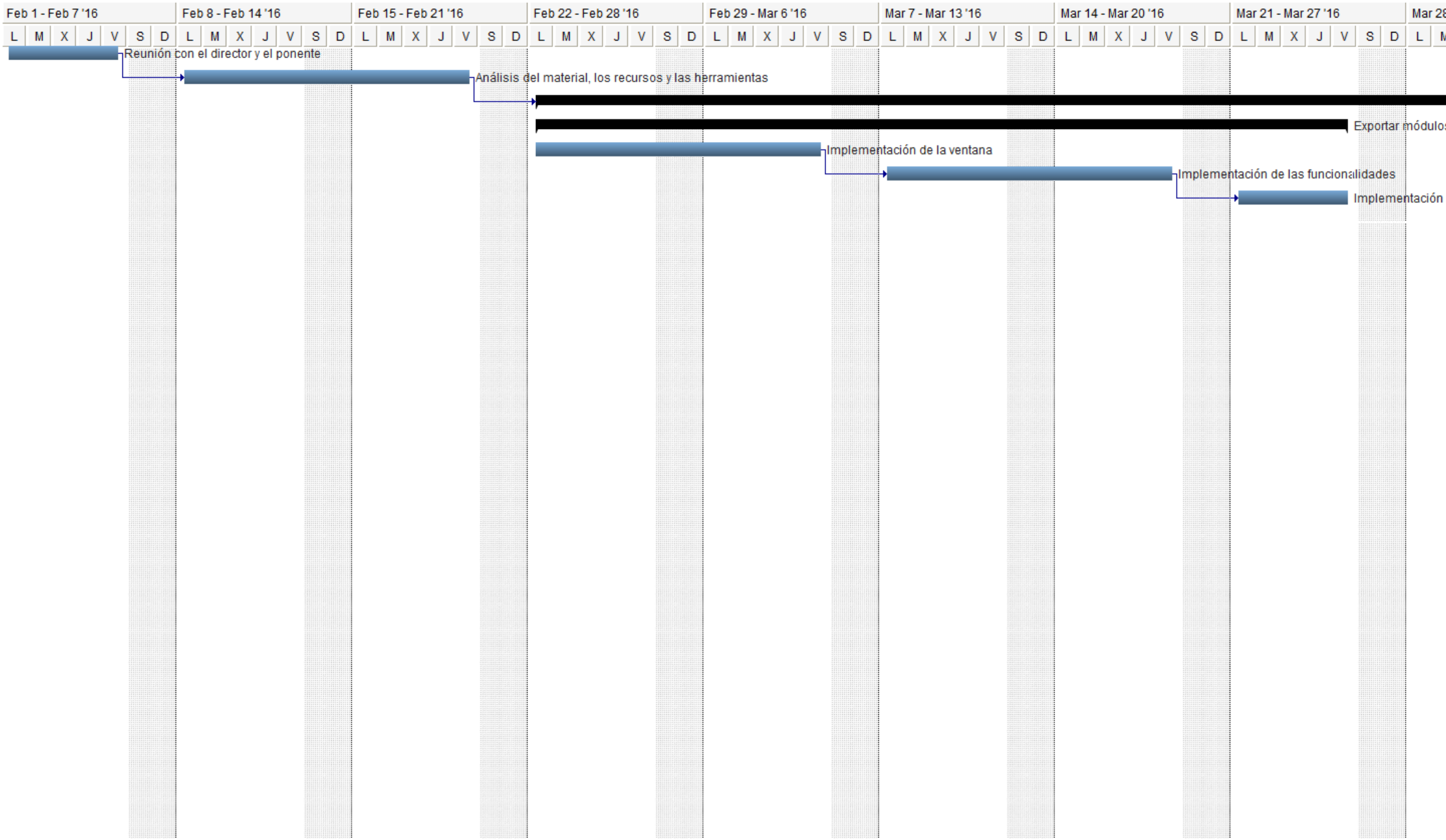
[9] Página principal de *Cloud9* - *https://c9.io/*

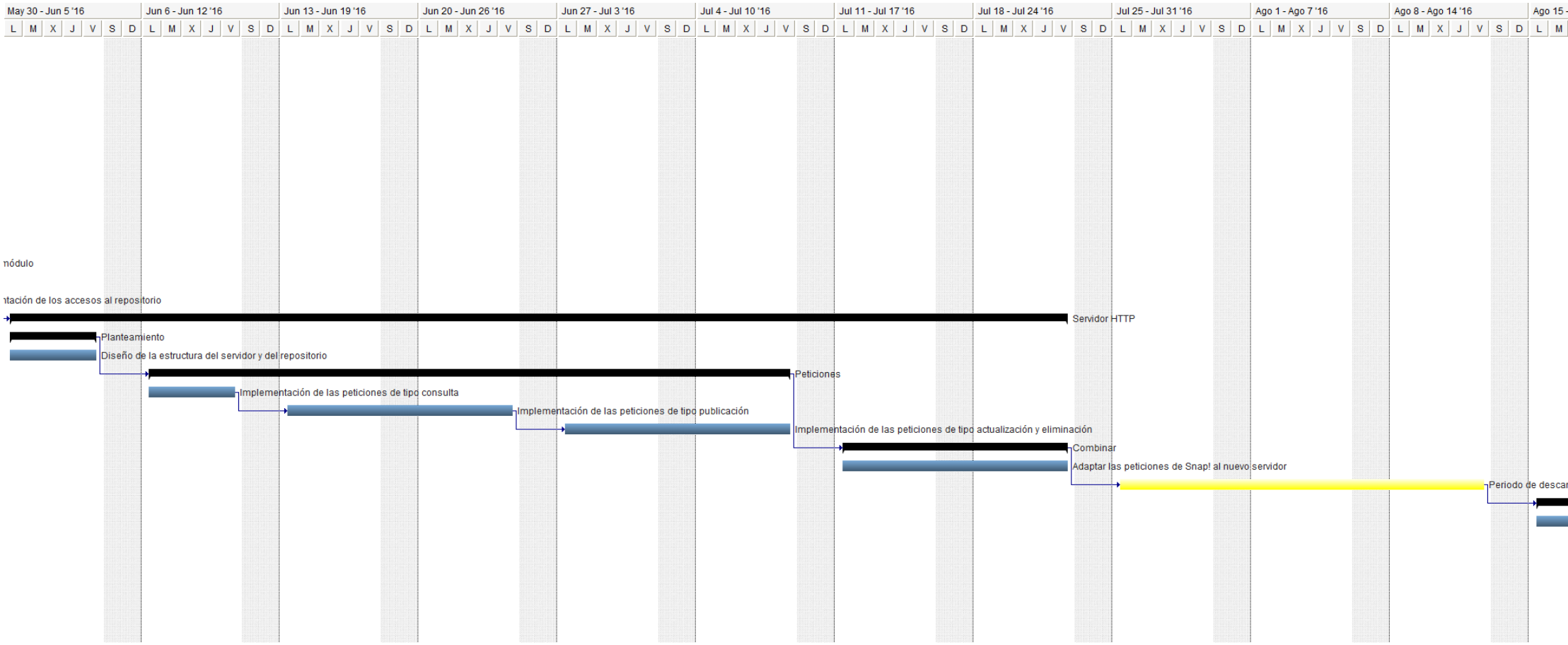
[10] Página principal de *Django* - *https://www.djangoproject.com/*

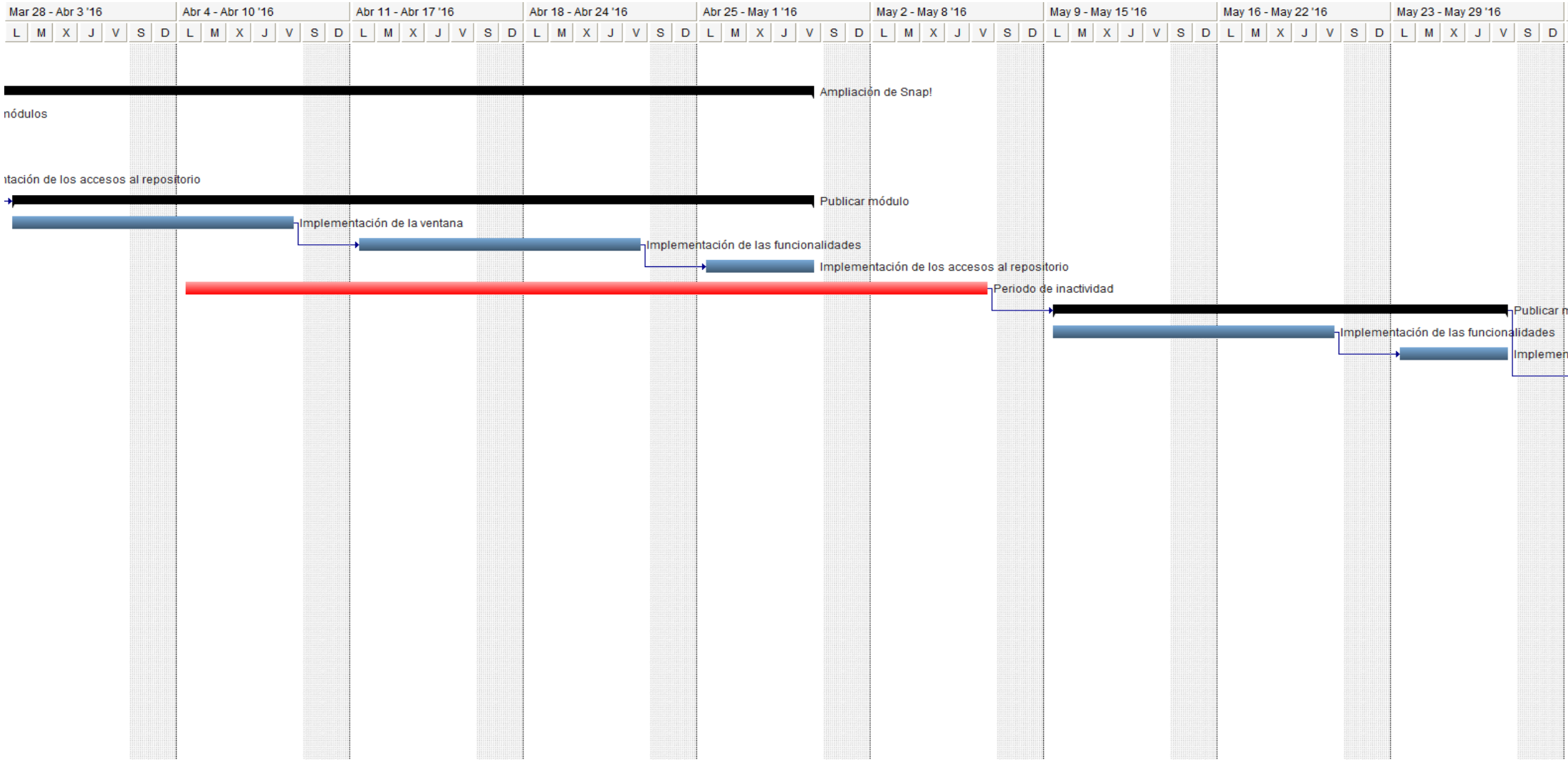
[11] Página principal de *Heroku* - *https://www.heroku.com/*

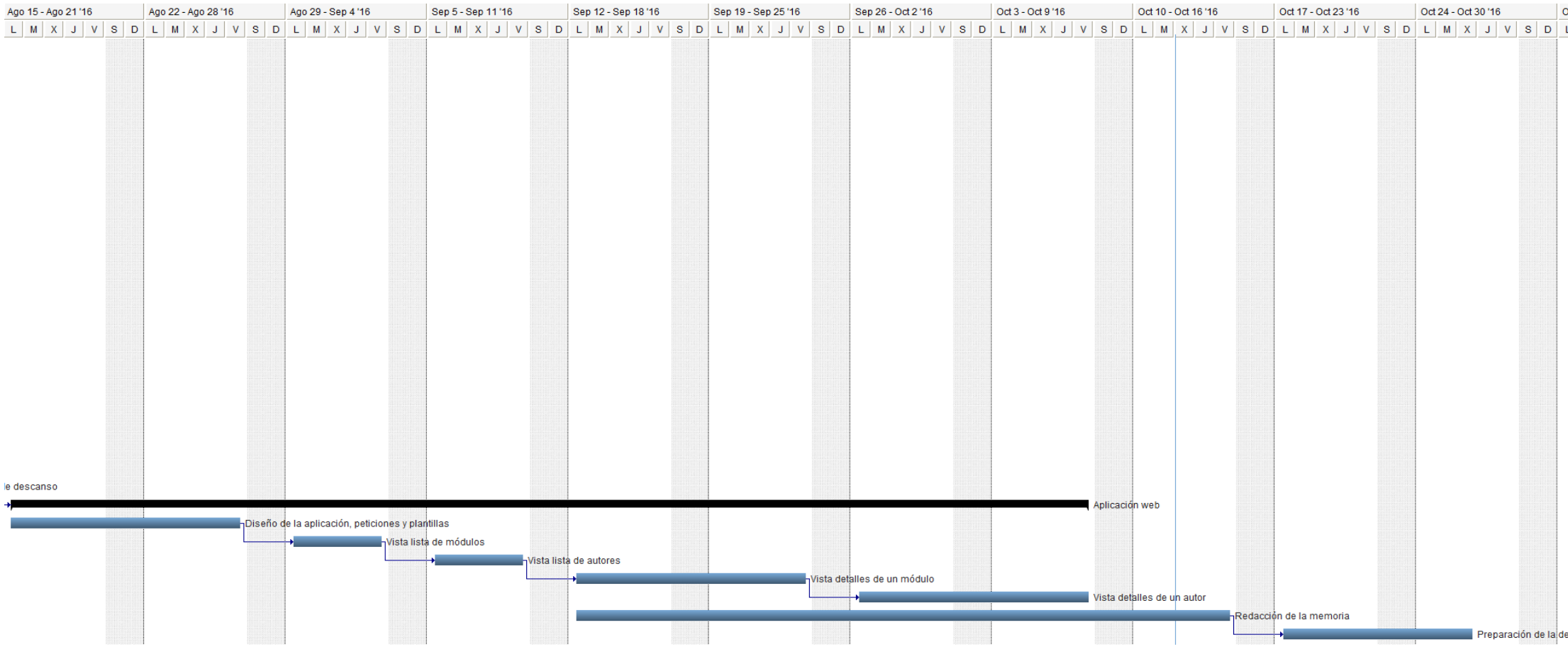
11 **Anexo**

11.1 Diagrama de *Gantt*


Feb 1 - Feb 7 '16
Feb 8 - Feb 14 '16
Feb 15 - Feb 21 '16
Feb 22 - Feb 28 '16
Feb 29 - Mar 6 '16
Mar 7 - Mar 13 '16
Mar 14 - Mar 20 '16
Mar 21 - Mar 27 '16
L M X J V S D
Reunión con el director y el ponente
Análisis del material, los recursos y las herramientas
Exportar módulos
Implementación de la ventana
Implementación de las funcionalidades
Implementación


May 30 - Jun 5 '16
Jun 6 - Jun 12 '16
Jun 13 - Jun 19 '16
Jun 20 - Jun 26 '16
Jun 27 - Jul 3 '16
Jul 4 - Jul 10 '16
Jul 11 - Jul 17 '16
Jul 18 - Jul 24 '16
Jul 25 - Jul 31 '16
Ago 1 - Ago 7 '16
Ago 8 - Ago 14 '16
Ago 15 -
L M X J V S D
nódulo
ntación de los accesos al repositorio
Servidor HTTP
Planteamiento
Diseño de la estructura del servidor y del repositorio
Peticiones
Implementación de las peticiones de tipo consulta
Implementación de las peticiones de tipo publicación
Implementación de las peticiones de tipo actualización y eliminación
Combinar
Adaptar las peticiones de Snap! al nuevo servidor
Periodo de descar


Mar 28 - Abr 3 '16
Abr 4 - Abr 10 '16
Abr 11 - Abr 17 '16
Abr 18 - Abr 24 '16
Abr 25 - May 1 '16
May 2 - May 8 '16
May 9 - May 15 '16
May 16 - May 22 '16
May 23 - May 29 '16
L M X J V S D
Ampliación de Snap!
nódulos
ıtación de los accesos al repositorio
Publicar módulo
Implementación de la ventana
Implementación de las funcionalidades
Implementación de los accesos al repositorio
Periodo de inactividad
Publicar m
Implementación de las funcionalidades
Implemen


Ago 15 - Ago 21 '16
Ago 22 - Ago 28 '16
Ago 29 - Sep 4 '16
Sep 5 - Sep 11 '16
Sep 12 - Sep 18 '16
Sep 19 - Sep 25 '16
Sep 26 - Oct 2 '16
Oct 3 - Oct 9 '16
Oct 10 - Oct 16 '16
Oct 17 - Oct 23 '16
Oct 24 - Oct 30 '16
L M X J V S D
le descanso
Aplicación web
Diseño de la aplicación, peticiones y plantillas
Vista lista de módulos
Vista lista de autores
Vista detalles de un módulo
Vista detalles de un autor
Redacción de la memoria
Preparación de la de